

## UTILITY PATENT APPLICATION TRANSMITTAL

(New Nonprovisional Applications Under 37 CFR § 1.53(b))

Attorney Docket No.  
GEMS8081.029

## TO THE ASSISTANT COMMISSIONER FOR PATENTS:

Transmitted herewith is the patent application of John V. Skinner, et al. entitled METHOD OF INTEGRATING X WINDOWS INTRINSICS BASED TOOLKITS AND WIDGETS WITH JAVA® for an:

(X) Original Patent Application. ✓

( ) Continuing Application (prior application not abandoned):

( ) Continuation ( ) Divisional ( ) Continuation-in-part (CIP)  
of prior application No: \_\_\_\_\_ Filed on: \_\_\_\_\_

( ) A statement claiming priority under 35 USC § 120 has been added to the specification.

Enclosed are 30 Total Pages. ✓

(X) Drawing(s); 3 Total Sheets. ✓

( ) Oath or Declaration:

( ) A Newly Executed Combined Declaration and Power of Attorney:

( ) Signed. ( ) Unsigned. ( ) Partially Signed.

( ) A Copy from a Prior Application for Continuation/Divisional (37 CFR § 1.63(d)).

( ) Incorporation by Reference. The entire disclosure of the prior application, from which a copy of the oath or declaration is supplied, is considered as being part of the disclosure of the accompanying application and is hereby incorporated herein by reference.

( ) Signed Statement Deleting Inventor(s) Named in the Prior Application. (37 CFR § 163(d)(2)).

( ) Power of Attorney.

(X) Return Receipt Postcard. ✓

( ) Associate Power of Attorney.

( ) A Check in the amount of \$ .00 for the Filing Fee.

( ) Preliminary Amendment.

(X) Information Disclosure Statement and Form PTO-1449. ✓

(X) A Duplicate Copy of this Form for Processing Fee Against Deposit Account. ✓

( ) A Certified Copy of Priority Documents (if foreign priority is claimed).

( ) Statement(s) of Status as a Small Entity.

( ) Statement(s) of Status as a Small Entity Filed in Prior Application, Status Still Proper and Desired.

( ) Other: \_\_\_\_\_

## CLAIMS AS FILED

FOR	NO. FILED	NO. EXTRA	RATE		FEE
			Large Entity	Small Entity	
Total Claims	37 ✓	17	\$18.00	\$ 9.00	\$306.00 ✓
Independent Claims	4 ✓	1	\$78.00	\$39.00	\$78.00 ✓
Multiple Dependent Claims (if applicable)					\$0.00
Assignment Recording Fee					\$0.00
Basic Filing Fee					\$690.00 ✓
Total Filing Fee					\$1,074.00 ✓

Charge \$ 1,074.00 to General Electric Company Deposit Account 07-0845 pursuant to 37 CFR § 1.25. At any time during the pendency of this application, please charge any fees required or credit any overpayment to this Deposit Account.

Respectfully submitted,

By: \_\_\_\_\_

Timothy J. Ziolkowski, Esq.  
Attorney of Record, Reg. No. 38,368

Date: 9/20/00

Correspondence Address:

BOYLE FREDRICKSON & ZIOLKOWSKI S.C.  
Suite 1030  
250 East Wisconsin Avenue  
Milwaukee, Wisconsin 53202  
Phone: (414) 225-1663  
Fax: (414) 225-9753

I hereby certify that this is being deposited with the U.S. Postal Service "Express Mail Post Office to Addressee" service under 37 CFR § 1.10 on the date indicated below and is addressed to:

Assistant Commissioner for Patents  
Box Patent Application  
Washington, D.C. 20231

By: Rosa Stong

Typed: Rosa Stong

Express Mail Label No.: EL600059926US

Date of Deposit: 9-20-00

09678207.092000

METHOD OF INTEGRATING X WINDOWS INTRINSICS  
BASED TOOLKITS AND WIDGETS WITH JAVA<sup>®</sup>

Inventors: John V. Skinner  
David P. Edwards

METHOD OF INTEGRATING X WINDOW INTRINSICS  
BASED TOOLKITS AND WIDGETS WITH JAVA®

CROSS-REFERENCE TO RELATED APPLICATION

This application claims the benefit of prior U.S. Provisional Application Serial No. 60/215,926, filed July 3, 2000 and entitled METHOD OF INTEGRATING X WINDOWS INTRINSICS BASED SOFTWARE AND JAVA.

5 BACKGROUND OF THE INVENTION

The present invention relates generally to using already developed software of one type with another type of software and more particularly to, a method of integrating preexisting X Windows Intrinsics based toolkits, widgets and the like with JAVA®.

JAVA® is a registered trademark of, and a software development system originated by, Sun Microsystems, Inc., of 901 San Antonio Road, Palo Alto, California 94303, that includes a programming language that is somewhat similar to C and includes extensive development tools along with unique technologies that permit software developers to develop software applications that can be run on many different kinds of computers and operating systems in an attempt to provide a platform-independent development system. Thus, with JAVA, a software developer does not have to include certain code that is specific for the type of computer on which the program will run. As a result, an application written using JAVA can run on different kinds of computers and with different kinds of operating systems. For example, JAVA has been designed so that the same application written in JAVA can be run, largely without any modification whatsoever, on computers that are as different as a PC, a Macintosh computer, a VMX computer, and a Unix computer.

JAVA is relatively young, having been introduced only recently in 1995. Although the number of applications written in JAVA is undoubtedly increasing every day due to its popularity, some other long established development systems have many tried-and-true building-block programs already at their disposal. In developing a JAVA application, it can be desirable to integrate or use at least some of this preexisting software with the application being developed to avoid reinventing the wheel. Using preexisting software advantageously saves time, because it does not need not be ported over to JAVA and debugged, saves money, because so much time is saved, and, in the end, can produce a more robust, feature-filled, and reliable JAVA application.

A visualization toolkit is an example of one type of software that can be integrated into JAVA for this purpose. A visualization toolkit typically includes a suite of simple, yet powerful, object components that are typically used in the development of programs that have two and three dimensional graphic and visualization capabilities as well as programs that handle certain types of input and output. For example, there are a number of visualization and graphics toolkits available for the X Window system, which is a graphics protocol and support system for building graphical user interfaces in Unix and Unix based systems. X Window visualization and graphics toolkits include MOTIF®, GIMP, and VTK, as well as other specialized third party toolkits.

MOTIF® is registered trademark of, and a graphical user interface standard that refers to a publicly licensed toolkit that was developed by, Open Software Foundation, Inc. of 11 Cambridge Center, Cambridge, Massachusetts 02142, which is also known as The Open Group of 29 Montvale Avenue, Woburn, Massachusetts 01801. GIMP is an open source toolkit that is also referred to as GTK or GTK+ and is available from



www.gtk.org. VTK is an X Window visualization toolkit that is available from Kitware at www.kitware.com.

5 A visualization and graphics toolkit common to X Window and third party toolkits is the X Window Intrinsics toolkit, which is also referred to as X Toolkit Intrinsics or Xt Intrinsics. This toolkit implements a set of user interface features or application environments that include object components, such as, menus, buttons, labels, scroll bars, scale-bars, drawing areas, and many other things. These object components are typically referred to as Widgets. There other types of Widgets that help manage things that are not necessarily graphical in nature, such as mouse and keyboard events. Using object-oriented techniques, an X Window based application manipulates Widgets provided by or made using the Xt Intrinsics toolkit or another higher level toolkit based on Xt Intrinsics, typically to do something on screen.

10 The Xt Intrinsics toolkit is also used by programmers to create new Widgets as well as new libraries or toolkits of Widgets. All of the aforementioned third party toolkits that exist and others are based on Xt Intrinsics and some of them are quite specialized. For example, there are toolkits used for processing on-screen graphics, producing 3-D graphics, animating objects, providing graphical analysis of data, and for many other applications.

20 Xt Intrinsics based visualization and graphics toolkits require an X event loop to provide behind the scenes support for communication between Widgets used by an application and an X Windows' X Server. JAVA does not directly provide any support for Xt Intrinsics and does not provide a method of supporting an X event loop. Past and present solutions to reusing Xt Intrinsics based visualization toolkits within a JAVA application or applet require that the application be separated into client and server

processes and have some form of interprocess communication. The client process, called an X Client, provides the basic application structure written in JAVA. The server process, called the X Server, contains the X Windows Intrinsics graphics services, including any desired Widgets, that are made available through interprocess communication. Unfortunately, this solution requires additional development time and effort to provide the interprocess communication and the additional complexity needed to manage the client and server states.

As the following example demonstrates, an invalid state can occur because this limited solution provides no special concurrency control to properly synchronize the X event loop with the JAVA application. Assume that the desired goal of at least a portion of the JAVA application is to cause a 3-dimensional scene to be displayed that rotates continuously at varying rates. Also assume that the native Xt Intrinsics visualization code receives an expose event caused by revealing a window used to display a 3-dimensional scene. This expose event will cause the current 3-dimensional scene to be redrawn. Concurrent to this, the JAVA application requests a schedule camera position change as the 3-dimensional scene is rotated. This will cause a concurrent update of the camera transforms by the JAVA application while the camera transforms are also being read by the X event loop. Unfortunately, the transforms will likely be in an invalid state for the draw that was prompted by the expose event, which can produce an error that halts or otherwise adversely effects execution of the JAVA application. This simple example serves to show that some form of concurrency control is needed for a JAVA application to be able to use preexisting visualization toolkits and components created with such toolkits.

The goal of the present invention is to be able to reuse software components that have already been developed for one type of platform or software development system, namely Xt Intrinsics or Xt Intrinsics based toolkits along with Widgets constructed therefrom, in a JAVA application.

5 It would therefore be desirable to have a method that makes integrating at least one preexisting Xt Intrinsics based software component into a JAVA application practical and reliable. It would also be advantageous to be able to integrate at least one such preexisting software component into a JAVA application using a method that properly synchronizes the JAVA application and the component. It would also be advantageous to be able provide a method that properly synchronizes the JAVA application and the preexisting software component in a manner that prevents data corruption due to concurrency between the software component and that portion of the JAVA application that calls the software component. It would also be advantageous to be able to provide a method that synchronizes a JAVA application and a preexisting Xt Intrinsics based visualization toolkit. It would also be advantageous to be able to provide a method that synchronizes a JAVA application and a preexisting Xt Intrinsics based visualization toolkit in a manner that prevents data corruption due to concurrency between the X event loop that services the toolkit and a thread or the threads of the JAVA application that make calls to the toolkit.

## SUMMARY OF THE INVENTION

The present invention provides a method of integrating preexisting software components, such as Widgets, that are included with and/or made using an X Windows Intrinsics or an X Windows Intrinsics based visualization toolkit, with a JAVA application.

In accordance with one aspect of the invention, a method of supporting concurrent X event processing and JAVA event processing is described. In this embodiment, a JAVA thread is devoted to executing an X event loop. An Xt Intrinsics file descriptor processing support function, XtAppAddInput, preferably is used as a means to suspend the X event loop to allow JAVA Native Interface (JNI) calls to be made from the JAVA application to the visualization toolkit. A method of acknowledgement is defined that assures that the toolkit's X event monitoring thread is safely suspended so that the JNI calls can be made without risk of data corruption due to concurrency between the X event processing thread and the JAVA application's calling thread and without concern for concurrent data updates to the toolkit data structures.

In accordance with another aspect of the invention, JAVA multithreading support services, such as JAVA's wait and notifyAll methods, are employed to allow safe conditions for multithreaded JAVA applications to also use the visualization toolkits, *i.e.*, to assure serial execution of JNI calls to the visualization toolkit.

In accordance with another aspect of the invention, when JAVA's Abstract Window Toolkit (AWT) is implemented using Xt Intrinsics and with multiple application threads, the use of a separate Xt Intrinsics Application Context to service the X event loop is employed. Windows are managed in separate hierarchies associated with each

application context. A specialized JAVA Canvas manages native windows transparent to the application programmer.

In accordance with still another aspect of the invention, a method for using an X Windows Intrinsics based visualization toolkit requiring an X event loop in a JAVA application is implemented that includes devoting a JAVA thread in the JAVA application to the X event loop; defining a write socket and a read acknowledge socket; calling the XtAppAddInput to register a process input function on the write socket, wherein the process input function performs a processInput and read, a write acknowledge, a read, and a return X event loop; defining a pausing method that performs a write on the write socket, and a read acknowledge on the read acknowledge socket; defining a resuming method that performs a write on the write socket; and when a call is made to the visualization toolkit, first calling the pausing method, making the call to the toolkit, and then calling the resuming method.

In accordance with a further aspect of the invention, a computer program is stored on computer readable storage media and includes a JAVA application thread that makes a call to a widget of an Xt Intrinsics based visualization toolkit and which has first and second write sockets; a JAVA process thread that creates an X event loop that has first and second read sockets and which can execute a blocking read that suspends further execution of the X event loop; wherein (1) a first data element is put on the first one of the write sockets by the JAVA application, (2) the first data element is read by the first one of the read sockets of the event loop, and (3) the blocking read is executed suspending further execution of the X event loop; wherein the call to the widget of the Xt Intrinsics based visualization toolkit is made; and wherein thereafter (1) a second data element is put on the second one of the write sockets by the JAVA application, (2) the

second data element is read by the second one of the read sockets of the event loop, and  
(3) the blocking read is completed causing the event loop to resume execution.

In accordance with a still further aspect of the invention, the JAVA application  
thread of the computer program further comprises a read socket and can execute a  
5 blocking read that suspends further execution of the JAVA application thread; wherein  
the blocking read of the JAVA application thread is executed after the first data element  
is put on the first one of the write sockets of the JAVA application thread, which  
suspends further execution of the JAVA application; wherein the event loop further  
comprises a write socket; and wherein a third data element is put on the write socket of  
the event loop after the first data has been read by the first one of the read sockets and the  
10 data element is read by the read socket of the JAVA application thread, unblocking the  
blocking read of the JAVA application thread, and resuming execution of the JAVA  
application thread.

In accordance with another aspect of the invention, the computer program  
15 includes a call to the XtAppAddInput function to register an input function that handles  
the first data element read from the first one of the read sockets.

Various other features, advantages and objects of the present invention will be  
made apparent from the following detailed description and the drawings.

## BRIEF DESCRIPTION OF THE DRAWINGS

The drawings illustrate the best mode presently contemplated for carrying out the invention.

In the drawings:

5 Fig. 1 is a schematic diagram that illustrates the typical hardware and software components of a computer running the X Window system;

Fig. 2 is a block diagram illustrating the typical elements of the X Window system;

Fig. 3 is a schematic diagram of one preferred general arrangement used in JAVA for integrating native or preexisting code with a JAVA application; and

Fig. 4 is a flowchart diagram depicting a preferred method of integrating an Xt Intrinsics based toolkit into a JAVA application.

## DETAILED DESCRIPTION OF AT LEAST ONE PREFERRED EMBODIMENT

Fig. 1 illustrates an overview of an X Window system 20, which is a system that provides a way of writing device independent graphical and windowing software that can easily be ported from one computer to another computer. To date, it has primarily been implemented on computers and computer platforms running Unix or a Unix based operating system, such as Linux. Fig. 2 illustrates additional software components that make up the X Window system 20.

Referring to Figs. 1 and 2, the X Window system 20 is made up of two primary components, an X Server 22 and an X Client 24. The X Server 22 is software that runs on a computer 26, such as a workstation, a PC, or the like, that is responsible for displaying windows and graphics on a display or a screen 28 of that particular computer. The X Server 22 has at its disposal, for example, graphics content 30, colormaps 32, fonts 34, cursors 36, and pixmaps 38 that are its building blocks. The X Server 22 interfaces with device drivers 40 to enable its output to be displayed on the monitor 28.

The X Client 24 includes software that can be run on the same computer 26 or a different computer, such as a network server or another workstation. The X Client sends messages to the X Server 22 when it needs to display graphics, text, windows, and other graphics related things as well as receive information about the occurrence of certain events, such as an event generated by a mouse 23 or a keyboard 25.

The computer 26 can also include storage media 29 on which software can be stored. Examples of storage media 29 include a hard drive, a removable disk or cartridge in a drive or connector that accommodates that kind of storage media, a flash memory device, or another kind of storage media device. The storage media 29 can be self-contained and disposed completely inside the computer 26, can be of removable



construction, and can be in a separate module that is linked to the computer 26, such as via a network or another means.

The X Client 24 includes X Lib 42, which is a low-level interface that provides communication paths, called connectors 44, 46, between it and the X Server 22. One connector 44 is used to communicate messages from X Lib 26 to the X Server 22 and the other connector 46 is used to communicate events, errors, and replies from the X Server 22 to the X Lib 26. The main task of the X Lib 26 is to translate data structures and procedures, usually written in C, into the messages that are communicated via connector 44 as well as to convert the messages received via connector 46 into C structures.

Referring to Fig. 2, the X Client 24 can also include a Client Application 48, a Window Manager 50, Xt Intrinsics 52, an Xt Intrinsics based toolkit, such as a MOTIF® Widget Set 54, and a JAVA Application 56. Not all of these are required. For example, in some instances a Client Application 48 is not required. In other instances, the MOTIF® toolkit 54 is not required. If desired, the X Client 24 can include more than one Client Application 48, more than one Window Manager 50, more than one Widget Set 54, or more than one JAVA Application 56.

Each Client Application 48 (sometimes also called “application client”) is an X Windows specific software program that is specifically designed to run under the X Windows system. Examples of a few known X Clients include bitmap, gs, olwm, MATLAB®, gnuplot, resize, scale, apE, idl, idraw, xauth, xclock, xcolors, xcalc, xconsole, xdb, xdbx, xcalc, xdm, xdtm, xdpr, xgif, xhost, xman, xbiff, xgraph, xeyes, and xterm, to name a few. There are many, many other such client applications. MATLAB is a math software package that is commercially marketed by MathWorks, Inc., of Cochituate Place 24 Prime Park Way, Natick, Massachusetts 01760.

The Window Manager 50 is a client application that is granted special permission by the X Server 22 to have supervisory powers over window requests from other applications. Any top-level window operation that an application requests must be verified by the Window Manager 50 before the operation will take place. The Window Manager 50 is free to ignore any request that does not conform to the graphical user interface maintained by the particular Window Manager 50. The Window Manager 50 also provides window decorations that give a windowing system its distinctive look and feel. The Window Manager 50 provides a means for a user to control the position, size, relative depth, and other parameters of one or more windows on a screen. Examples of Windows Managers include twm, mwm, olwm, and DECWindows.

As already discussed above, Xt Intrinsics includes toolkit parts that allow programmers to build new Widgets as well as new libraries, sets, or toolkits of Widgets. The MOTIF<sup>®</sup> Widget Set 54 is an example of one such Widget set. There are many other widget sets, toolkits and libraries available. Examples of toolkits include Athena, VTK, OpenGL, and GIMP. VTK is an example of a preferred toolkit that is used for 2D and 3D graphics and visualization applications that have uses in medical and industrial fields.

VTK is an example of a preferred visualization toolkit because it forms the basis of many custom third party toolkits used for many graphics and visualization applications. For example, GE Medical System's visualization toolkit is called GVTk and is a toolkit that is based on VTK. Examples of graphics and visualization applications for which these toolkits are well suited include medical volume rendering and imaging, interactive engine visualization, financial data visualization, visualization of results of engineering analyses, reconstruction of anatomical structures of patients from CT, MRI and ultrasound scans, and virtual generation of printed matter, electronic

manuals, and training materials. One machine on which a method encompassing the present invention can be implemented is a Sun Ultra workstation using a SunOS 5.5 or SunOS 5.6 operating system.

The JAVA Application 56 is shown in Fig. 2 as interfacing with X Lib 42, Xt  
5 Intrinsics 52, and a MOTIF<sup>®</sup> Widget Set 54. If desired, the JAVA Application 56 can interface with toolkits other than MOTIF<sup>®</sup>. For example, the JAVA Application 56 can interface with the VTK toolkit in addition to or in lieu of MOTIF<sup>®</sup>. The JAVA Application 56 can also interface with toolkits that are based on the VTK toolkit and which are used for graphics and visualization applications, such as one or more of those described above.

Fig. 3 is a diagram that generally illustrates how a JAVA Application 56 can use  
already existing native applications and native libraries, *e.g.* toolkits, written in other languages, including the C or C++ language of Xt Intrinsics. For example, the JAVA Native Interface (JNI) 58 interfaces with JAVA classes 60, a JAVA virtual machine (VM) 62, and exceptions 64, if any are to be taken into account. The JNI 58 also  
15 interfaces with one or more native functions 66 and one or more native libraries 68. The JNI 58 can also be used to interface native routines, native classes, native debuggers, and native typecheckers with a JAVA application.

Fig. 4 illustrates a preferred method for using Xt Intrinsics or an Xt Intrinsics  
20 based toolkit with a JAVA application 56. Although JAVA does not provide direct support for Xt Intrinsics, the X Event Loop 70 required to support communication between Xt Intrinsic based Widgets and the X Server preferably is implemented in C and called via the JAVA application 56 by a process thread called X Event Loop Thread 72 that is created using the JAVA Thread class. The JAVA application 56 also includes at

that is created using the JAVA Thread class. The JAVA application 56 also includes at least one JAVA application thread, such as JAVA Application Thread #1 74, that contains application code and which communicates with the X Event Loop 70 to provide concurrency control. Where the application 56 has more than one application thread, each application thread that makes calls to visualization toolkits, individual Widgets, and the like is also configured to provide concurrency control.

The X Event Loop 70 provides event monitoring for the Xt Intrinsics based toolkit that will be accessed by it and the application thread 74. The X Event Loop 70 has some special supporting resources. First, a pair of sockets 76, 78 are created with one of the sockets being a write socket, referred to in Fig. 4 as write 76, in the application thread 74 and the other one of the sockets being a read socket, referred to in Fig. 4 as read 78, in the X Event Loop 70. Both of these sockets 76, 78 are connected or linked in such a way that they are essentially piped together so that output of the write socket 76 is automatically directed to the read socket 78.

Second, a callback function, identified as processInput 78, is registered with Xt Intrinsics to process data delivered to the read socket 76. The processInput callback function relies on the standard Xt Intrinsics XtAppAddInput function to control thread execution of the X Event Loop. XtAppAddInput is a file descriptor that is used to register a new source of events with Xt Intrinsics. In the present method, the XtAppAddInput is used to register the processInput function 78 preferably on the write socket 80.

Third, another pair of sockets, called write acknowledge 80 and read acknowledge 82, are created. Fourth, an application context (not depicted in Fig. 4) is created for the X Event Loop 70. Once these resources are created, the X Event Loop Thread 72 is created

in JAVA. This thread 72 calls a special native code function, typically written in C, that serves as the X Event Loop 70. This thread 72 remains in execution the entire time that the JAVA application is running.

Two special functions, `pauseEventLoop` 84 and `resumeEventLoop` 86, are implemented in the application thread 74 to provide concurrency control by preventing concurrency conditions from arising so that both the JAVA application 56 and the Xt Intrinsic 52 (and toolkit 54) run smoothly without interfering with each other and with locations and objects, such as Widgets, to which they both might attempt to send data or access. These two functions 84, 86 accomplish this by holding the X Event Loop Thread 72 in a safe location while the JAVA application thread 74 makes a visualization toolkit call 88 to Xt Intrinsic 52 or an Xt Intrinsic based toolkit 54. Preferably, these functions do so by selectively suspending execution of the X Event Loop 70 while the visualization toolkit call 88 is made and then resuming execution of the X Event Loop 70 after the call has been made.

During operation of the application thread 74 and while the X Event Loop 70 is running concurrently, the `pauseEventLoop` 84 function writes a first data element 92 to the write socket 76. The data element 92 can be a token data element, such as a simple byte or a token. The `pauseEventLoop` 84 then performs a blocking read on the read acknowledge socket 82, which pauses execution of the application thread 74 until a second data element 94 is subsequently received by the read acknowledge socket 82. The X Event Loop Thread 72 and, hence, the X Event Loop 70, continue execution until the first data element 92 is discovered on the write socket 76. For example, depending at what point the X Event Loop is at in its execution, the X Event Loop 70 can completely

finish processing its current X event and then return to check for a data element on its read socket 78.

When the first data element 92 is discovered on the write socket 76, the X Event Loop 70 invokes the processInput function 78, reads the data element 92 from the write socket 76, writes a second data element 94 to the write acknowledge socket 80, and performs a blocking read on a second read socket 90 of the loop 70. The data element 94 written to the write acknowledge socket 80 is directed to the read acknowledge socket 82 of the pauseEventLoop 84 of the application thread 74 causing the read acknowledge block to unblock. When unblocked, execution of the application thread 74 resumes and control preferably is transferred to the thread 74.

At the same time or about the same time that control is returned to the application thread 74, the blocking read performed on the second read socket of the X Event Loop 70 causes the loop 70 to pause execution. This pause in execution of the X Event Loop 70 permits a visual toolkit call 88 to be made to a graphics/visualization object that can be a toolkit 52 or 54 or a Widget. Although not depicted in Fig. 4, this call 88 is made using the JAVA Native Interface 58. As a result of suspending execution of the X Event Loop 70, the call 88 can safely be made with any fear of concurrency related data corruption. Selectively suspending execution of the X Event Loop 70 to permit a visualization toolkit call 88 to be made provides concurrency control that advantageously prevents concurrency related data corruption.

Once the visualization toolkit call 88 is made, the application thread calls the resumeEventLoop function 86. Calling this function 86 writes a third data element 96 to a second write socket 98 of the application thread 74 and continues execution of the thread 74. When the third data element 96 is discovered on the second read socket 90 of

processInput 78 of the X Event Loop 70, it unblocks the second read 90 and causes the X Event Loop 70 to resume execution.

Where the JAVA application has multiple threads, such as JAVA Application Thread #1, identified in Fig. 4 by reference numeral 74, JAVA Application Thread #2, identified by reference numeral 100, and JAVA Application Thread #3, identified by reference numeral 102, additional concurrency control is needed so that only one application thread at a time can make a visualization toolkit call 88. To provide this control, a multithreaded application has a special JAVA class, called JContext class, that contains two JAVA methods, jPauseEventLoop 104 and jResumeEventLoop 106, that mirror the pauseEventLoop and resumeEventLoop functions. These methods 104, 106 prevent more than one application thread from invoking pauseEventLoop 84 and making a visual toolkit call 88 through the use of a member variable of this class that each application thread has that is called inuse. One JContext object 120 is created in the JAVA application that includes inuse, jPauseEventLoop 104, and jResumeEventLoop 106 to synchronize the threads 74, 100 and 102 and control access to the visualization toolkit(s), Widgets, etc. by all threads 74, 100, and 102 of the application.

The member variable inuse keeps track of the condition of when a pauseEventLoop 84 has been called but a matching variable resumeEventLoop 86 in the same application thread has not been called. The member variable inuse indicates this very condition when it is set to USED 108. When inuse is set to UNUSED 110, a new pauseEventLoop 84 can be called.

One of the JAVA methods, jPauseEventLoop 104, is a mirror of pauseEventLoop 84. This method 104 waits until inuse is set to UNUSED 110. If inuse is set to UNUSED at the time when thread execution reaches jPauseEventLoop 104,

jPauseEventLoop 104 sets it to USED 110 and calls pauseEventLoop 84. However, if inuse is already set to USED when thread execution reaches jPauseEventLoop 104, pauseEventLoop 84 will call the standard JAVA Object wait() method 112 and execution of that application thread will be suspended until another application thread calls the standard JAVA Object method notifyAll 114. This method 114 is present in each application thread that makes a visualization toolkit call.

The mirror of resumeEventLoop 86, called jResumeEventLoop 106, calls resumeEventLoop 86. Once control returns from resumeEventLoop 86, inuse is set to UNUSED 110 and the notifyAll 114 is invoked. This triggers pending jPauseEventLoop calls to each of the threads to check their inuse variable. Ultimately, this check of inuse of each thread will permit jPauseEventLoop 104 of one of the application threads to gain access to pauseEventLoop 84 of that thread and cause jPauseEventLoop 104 of every other application thread to execute a call to wait 112 that suspends their execution.

Access to inuse is controlled through JAVA's synchronized method support used by jPauseEventLoop 104 and jResumeEventLoop 106. Rather than the application thread invoking pauseEventLoop 84 and resumeEventLoop 86, the application thread invokes jPauseEventLoop 104 and jResumeEventLoop 106, which serve to thereby assure the sequential execution of pauseEventLoop 84, the JNI visualization toolkit call 88, and resumeEventLoop 86 across multiple application threads.

In at least one preferred implementation of the method of this invention, the method requires two separate application contexts, with each one associated with one or more X windows. For example, the JAVA Abstract Window Toolkit (AWT) application context will typically be associated with one or more top level windows into which native drawing is done, such as through calls made to visualization toolkits, Widgets, and other



threads 74, 100, 102, which owns a native top-level window. The native top level window has no decoration and is, in effect, “pasted” onto the resultant canvas so that any expose, size, move, minimize, maximize, raise, lower, and/or other operation on the canvas also causes analogous operations on the native window. This allows JAVA

5 programmers to simply work with canvas objects as they normally would and not have to worry about these other details. It should be noted that the native window and the canvas belong to completely separate window hierarchies, each associated with their respective application contexts. For example, the Swing implementation provides a NativeJPanel, which is analogous to the NativeCanvas class.

10 In operation of a JAVA application having multiple threads, such as the application 56 depicted in Fig. 4, only one thread 74 gains access to change the value of inuse to USED 108 from jPauseEventLoop 104. The remaining threads 100, 102 are suspended by a JAVA wait call 112. The surviving thread calls pauseEventLoop 84 and pauseEventLoop 84 performs a write on the write socket 76 to cause the processInput

15 function 78 to be invoked by the X Event Loop 70. The processInput function 78 reads off the data element or token 92 from the read socket 78, writes a token 94 back on its write acknowledgment socket 80, and starts a blocking read in its read socket 90, which causes the X Event Loop 70 to suspend until resumeEventLoop 86 of the application thread 74 writes to its write socket 98.

20 After or concurrent with the suspension of the X Event Loop 70 on the blocking read 90, a call is made from the application thread 74 using JNI to the visualization toolkit 88. The JAVA method jResumeEventLoop 106 is then invoked to allow the X Event Loop 70 and the X Event Loop Thread 72 to resume execution. When invoked, jResumeEventLoop 106 writes a token 96 on its write socket 98 that satisfies the read 90,

Event Loop 70 and the X Event Loop Thread 72 to resume execution. When invoked, jResumeEventLoop 106 writes a token 96 on its write socket 98 that satisfies the read 90, which causes it to unblock and resume execution of the X Event Loop 72. After that, jResumeEventLoop 106 sets inuse to UNUSED 110, which can allow one of the JAVA application threads currently suspended on a wait call 112 to gain access to jPauseEventLoop 84 to perform another sequence of calls that can include other visualization toolkit calls. Thereafter, jResumeEventLoop 106 calls notifyAll 114 to awaken the threads 100, 102 that were previously suspended by a wait call 112.

The steps of the above-described method are preferably repeated at least once. Preferably, they are repeated as necessary until execution of the application stops or is stopped, such as by a user exiting the application.

The present invention has been described in terms of the preferred embodiment, and it is recognized that equivalents, alternatives, and modifications, aside from those expressly stated, are possible and within the scope of the appending claims.

## CLAIMS

1. A method of integrating an X Window visualization toolkit with a JAVA application comprising:

providing a JAVA application thread that includes a call to an X Window visualization toolkit and a JAVA process thread that comprises an X Window X event loop; and

suspending execution of the X event loop to prevent concurrency related data corruption while a call to the X Window visualization toolkit is made by the JAVA application thread.

2. The method of claim 1 wherein the X event loop comprises an X Window file descriptor function that coordinates an X event loop blocking read that is used to suspend execution of the X event loop.

3. The method of claim 1 wherein the application thread comprises a first write socket that is used to communicate a first data element to a first read socket of an XtAppAddInput file descriptor function of the X event loop, a read acknowledge socket of the application thread that acknowledges receipt of a second data element from a write acknowledge socket of the XtAppAddInput file descriptor function of the X event loop, and a second write socket that is used to communicate a third data element to a second read socket of the X event loop; and further comprising the steps of:

communicating a first data element from the first write socket of the JAVA application thread to the first read socket of the X event loop;

performing an application thread blocking read that suspends execution of the application thread until a second data element is discovered by the read acknowledge socket of the application thread;

communicating the second data element from the write acknowledge socket of the X event loop to the read acknowledge socket of the application thread;

resuming execution of the application thread when the second data element is discovered by the read acknowledge socket of the application thread;  
performing the X event loop blocking read that suspends execution of the X event loop until a third data element is discovered by the second read socket of the X event loop;  
20 making a call from the JAVA application thread to the visualization toolkit while execution of the X event loop is suspended;  
communicating the third data element from the second write socket of the application to the second read socket of the X event loop; and  
25 resuming execution of the X event loop when the third data element is discovered by the second read socket of the X event loop.

4. The method of claim 3 wherein the application thread further comprises a plurality of JAVA functions with a first one of the functions pausing execution of the X event loop by causing the first data element to be communicated to the first read socket of the X event loop and a second one of the functions resuming execution of the X event loop by causing the third data element to be communicated to the second read socket of the X event loop.

5. The method of claim 4 wherein the JAVA application comprises a plurality of application threads and a class defining a pair of JAVA methods having a first one of the JAVA methods that mirrors the first one of the functions that pauses execution of the X event loop for each application thread and a second one of the JAVA methods mirroring the second one of the functions that resumes execution of the X event loop for each application thread wherein the first and second JAVA methods prevent more than one application thread at a time from making a call to the visualization toolkit or widget.

5

5

5

5

5

5

12. The method of claim 10 wherein the member variable is set to the second one of the values before completing execution of the one of the application threads.

13. The method of claim 12 wherein a call is made to the standard JAVA Object notifyAll upon completion of the one of the application threads.

14. The method of claim 12 wherein making a call to notifyAll upon completion of the one of the application threads invokes the first one of the JAVA methods to determine which one of the plurality of application threads should be executed.

15. The method of claim 1 wherein the toolkit comprises at least one widget.

16. The method of claim 15 wherein the at least one widget comprises a visualization/graphics object.

17. The method of claim 1 wherein the JAVA application comprises a JAVA applet.

18. A method of using an X Window visualization toolkit or widget in a JAVA application or applet comprising:

(a) providing a plurality of JAVA application threads that each include a call to an X Window visualization toolkit or widget and a JAVA process thread that comprises an X Window X event loop;

(b) selecting one of the plurality of application threads to execute and then suspending execution of the remainder of the plurality of application threads;

(c) suspending execution of the X event loop while a call to the X Window visualization toolkit or widget is made by the one of the plurality of application threads;

(d) making a call to the X Window visualization toolkit or widget; and

(e) resuming execution of the X event loop.

19. The method of claim 18 comprising repeating steps (b) through (e) at least once.

20. The method of claim 19 wherein the X event loop performs a blocking read to suspend execution of the X event loop in step (c).

21. The method of claim 20 wherein the X event loop comprises an X Window file descriptor function that performs the blocking read.

22. The method of claim 21 wherein the X Window file descriptor function comprises the XtAppAddInput function.

23. The method of claim 18 wherein after step (b), an additional step comprising suspending execution of the one of the plurality of the application threads to allow the X event loop to finish processing any X event being processed by the X event loop before execution of the X even loop is suspended in step (c).

24. The method of claim 23 wherein suspending execution of the one of the plurality of the application threads is accomplished by a blocking read that includes a read socket of the one of the plurality of application threads that receives a data element from a write socket of the X event loop.

25. The method of claim 18 wherein suspending execution of the X event loop in step (c) comprises:

(1) providing a pause function of the one of the plurality of the application threads that communicates a first data element to a first write socket that is linked to a first read socket of the X event loop and a process data element function of the X event loop that reads the first data element from the read socket and issues a blocking read;

(2) communicating the first data element to the first write socket of the one of the plurality of the application threads;

- 10 (3) retrieving the first data element by the first read socket of the X event loop; and
- (4) invoking a blocking read suspending execution of the X event loop.

26. The method of claim 25 wherein resuming execution of the X event loop in step (e) comprises:

- 5 (1) providing a resume function of the one of the plurality of the application threads that communicates a second data element to a second write socket that is linked to a second read socket of the X event loop;
- (2) communicating the second data element to the second write socket of the one of the plurality of the application threads;
- (3) retrieving the second data element by the second read socket of the X event loop; and
- 10 (4) unblocking the blocking read resuming execution of the X event loop.

27. The method of claim 18 wherein suspending execution of the X event loop in step (c) comprises:

- 5 (1) providing a pause function of the one of the plurality of the application threads that communicates a first data element to a first write socket that is linked to a first read socket of the X event loop, and a process data input function of the X event loop that reads the first data element from the read socket and communicates a second data element to write acknowledge socket that is linked to a read acknowledge socket of the one of the plurality of the application threads;
- (2) communicating the first data element to the first write socket of the one of the plurality of the application threads;
- 10 (3) invoking a first blocking read suspending execution of the one of the plurality of the application threads;



15

20

(7) unblocking the first blocking read resuming execution of the one of the plurality of the application threads.

29. The method of claim 27 wherein the process data input function comprises the XtAppAddInput function native to the X Window system.

31. The method of claim 18 wherein the call to the X Window visualization toolkit or widget comprises a call to an X Window Intrinsics based toolkit or widget.

33. The method of claim 18 wherein the call to the X Window visualization toolkit or widget is made using the JAVA Native Interface.

- devoting a JAVA thread in the JAVA application to the X event loop;
- defining a write socket and a read acknowledge socket;

defining a pausing method that performs a write on the write socket, and a  
edge on the read acknowledge socket;

defining a resuming method that performs a write on the write socket; and when a call is made to the visualization toolkit, first calling the pausing

(a) a JAVA application thread that makes a call to a widget of an Xt and visualization toolkit and which has first and second write sockets;

(b) a JAVA process thread that creates an X event loop that has first and

ockets and which can execute a blocking read that suspends further

(c) wherein (1) a first data element is put on the first one of the write  
JAVA application, (2) the first data element is read by the first one of the

(d) wherein the call to the widget of the Xt Intrinsics based visualization

(e) wherein thereafter (1) a second data element is put on the second one-way communication channel by the JAVA application, (2) the second data element is read by the

second one of the read sockets of the event loop, and (3) the blocking read is unblocked causing the event loop to resume execution.

36. The computer program of claim 35 wherein:

(a) the JAVA application thread further comprises a read socket and can execute a blocking read that suspends further execution of the JAVA application thread;

5 (b) the blocking read of the JAVA application thread is executed after the first data element is put on the first one of the write sockets by the JAVA application thread, which suspends further execution of the JAVA application;

(c) the event loop further comprises a write socket; and

10 (d) a third data element is put on the write socket of the event loop after the first data has been read by the first one of the read sockets and the data element is read by the read socket of the JAVA application thread, unblocking the blocking read of the JAVA application thread, and resuming execution of the JAVA application thread.

37. The computer program of claim 35 wherein the XtAppAddInput function is called to register an input function that handles the first data element read from the first one of the read sockets.

# METHOD OF INTEGRATING X WINDOW INTRINSICS BASED TOOLKITS AND WIDGETS WITH JAVA®

## ABSTRACT

A method of integrating an Xt Intrinsics based toolkit with a JAVA application. The application includes a process thread that implements an X event loop and an application thread that suspends execution of the event loop to allow a call to be made through the JAVA Native Interface to a toolkit or a widget to cause something to be displayed. The application thread includes a write socket to communicate a token to a read socket of the event loop, and performs a blocking read suspending the thread. The event loop returns an acknowledgment token via a write socket to a read socket of the thread that unblocks its blocking read and the event loop invokes its own blocking read suspending its execution. The file descriptor function, XtAppAddInput, preferably, is used to read the token and to set the blocking read. After a toolkit call is made, a token is sent from a second write socket of the thread to a second read socket of the event loop that unblocks its blocking read resuming the event loop. JAVA multithreading support services, such as JAVA's wait and notifyAll methods, are employed in multithreaded applications to ensure that only one thread at a time can make a call. Use of a separate Intrinsics application context to service the event loop is employed, windows are managed in separate hierarchies associated with each application context, and a special JAVA Canvas manages native windows transparent to the programmer.

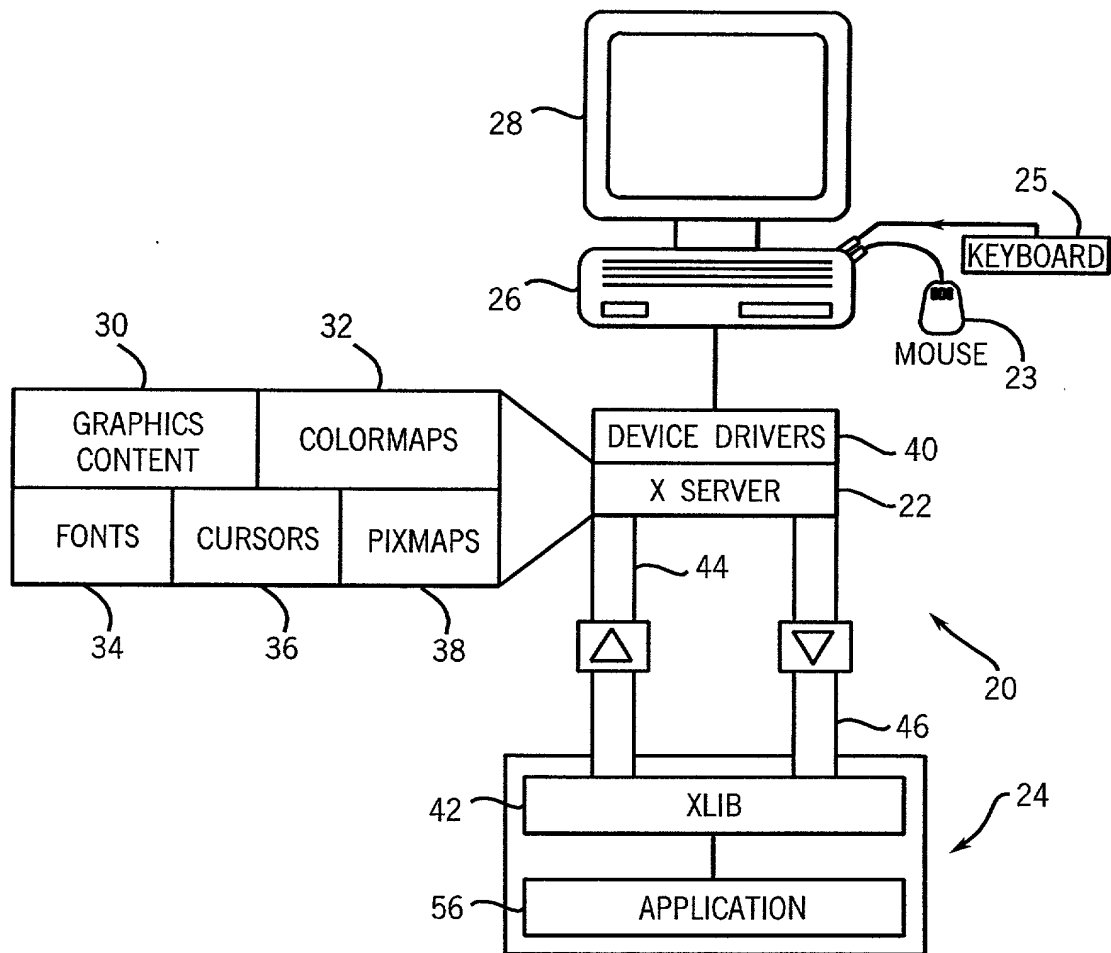


FIG. 1

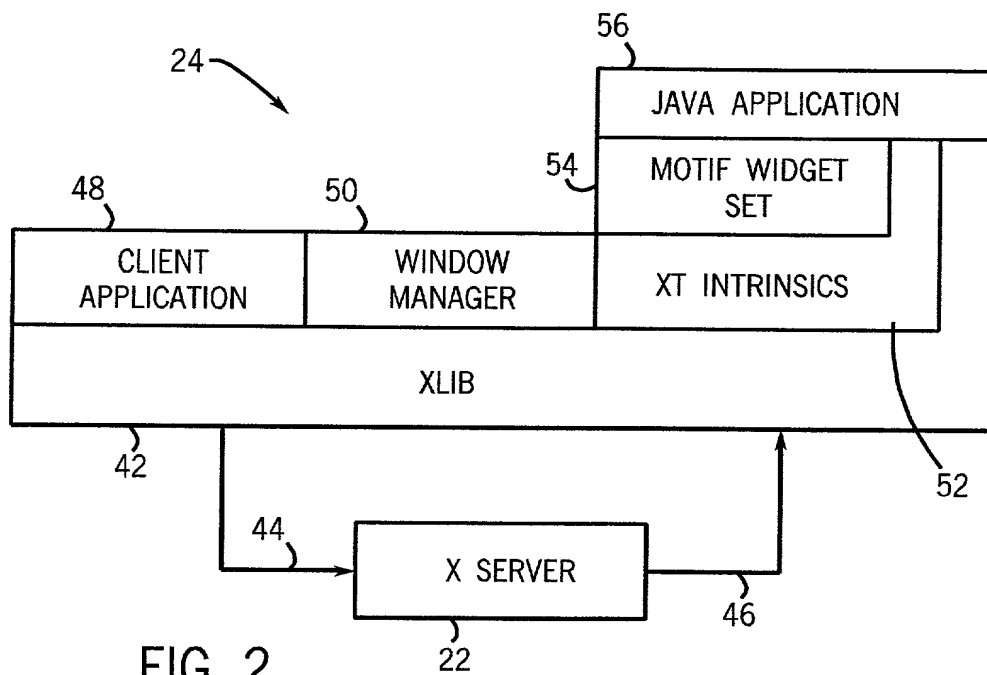


FIG. 2

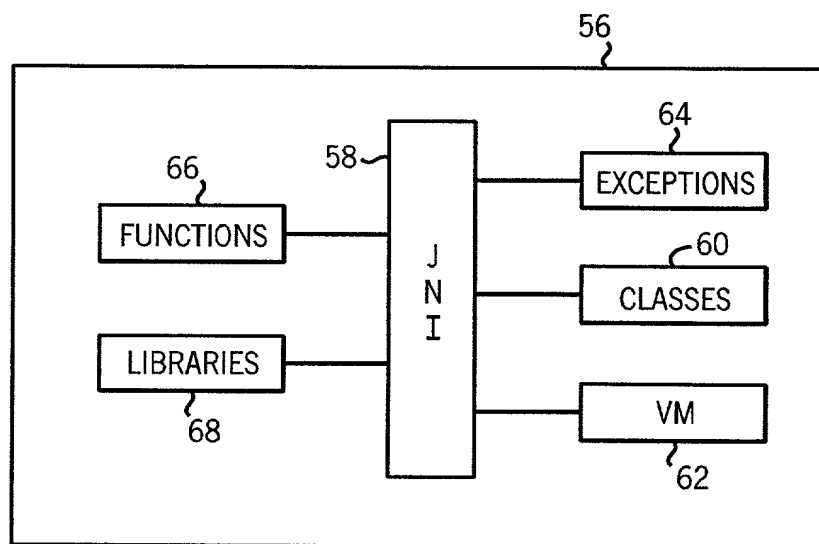


FIG. 3

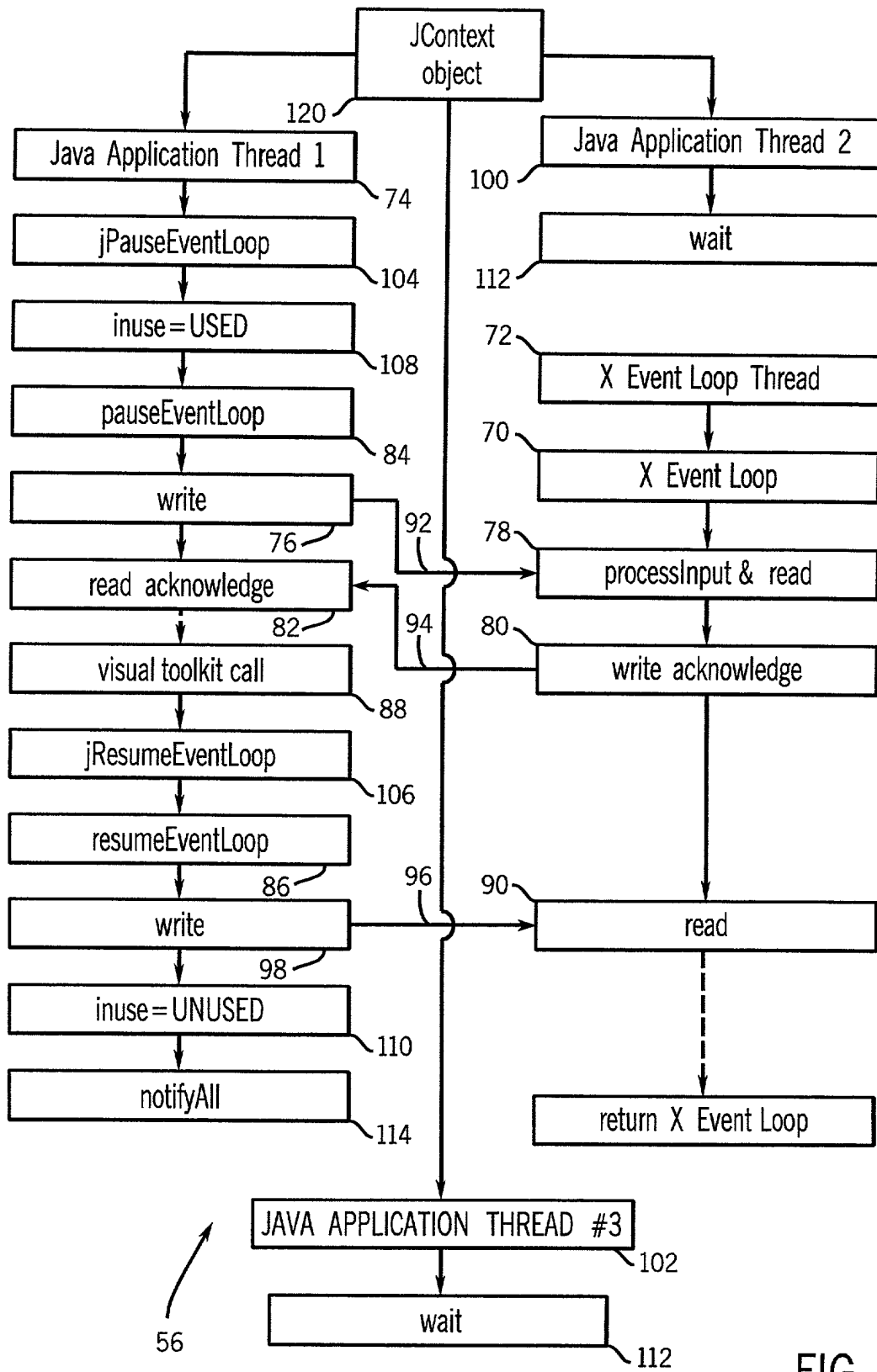
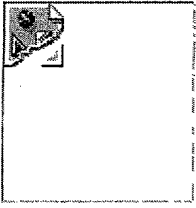


FIG. 4

[Next](#) [Up](#) [Previous](#)

# APPENDIX A



## X WINDOWS

Dr A D Marshall  
Room M 1.38

Copyright David Marshall 1994-97

---

## Prototype X Reference Material



[X Window/Motif Reference Demo](#)

---

---

## Search for Keywords in X Windows Notes

[Keyword Searcher](#)

- [Postscript Printable](#) versions of the lecture notes --- Local Students ONLY.
  - [Text Printable](#) versions of the lecture notes --- Local Students ONLY.
- 

## Lecture Schedule

[Week by Week Lecture Schedule](#)

---

---

## X COURSEWARE



Lecture notes etc.

# The Road to X/Motif

This book introduces the fundamentals of Motif programming and addresses wider issues concerning the X Window system. The aim of this book is to provide a practical introduction to writing Motif programs. The key principles of Motif programming are always supported by example programs.

The X Window system is very large and this book does not attempt to detail every aspect of either X or Motif. This book is not intended to be a complete reference on the subject.

The book is organised into logical parts, it begins by introducing the X Window system and Motif and goes on to study individual components in detail in specific Chapters. In the remainder of this Chapter we concentrate on why Motif and related areas are important and give a brief history of the development of Motif.

## Why Learn X Window and Motif?

There are many reasons and advantages to programming in X/Motif. A few of these are listed below:

- Motif provides an introduction to graphic user interface (GUI) programming -- all computers now employ some form of a GUI to their operating systems and other key applications. Most GUIs adhere to similar design principles. Motif can be regarded a high level GUI toolkit that adopts and enforces common GUI design principles.
- X Window provides a consistent means of graphical user interaction for UNIX workstations.
- Motif provides a high level toolkit, that already has many fully featured GUI objects. For example cut and paste, multi-line *text editors*, *file browsers*, *drag and drop* mechanisms. Simple yet usable Motif applications can be assembled by *bolting* such objects together. Motif *speeds* up GUI program development.
- The X Window system is device independent -- it can run on most common computer platforms. If there is a need for different platforms to interact together over a network, X Window might be a good way to achieve this.
- You may have been using the X Window system and want to understand how the system works.
- Professional X Window programmers are still not that numerous even though they are in great demand. You may be reading this book because you need to learn Motif for this reason. A quick scan through any Computer Vacancies Column in major Computer magazines, journals or employment agencies should highlight this need.

## How to use this book

### About this book

The X Window system, or simply X, is very large. It has been through many different versions since its conception although things are now becoming fairly standardised and established.

This book does not attempt to cover all of the X system. We will only look at important parts of the system. Indeed we only study important parts of Motif which is itself a component (albeit a large and significant one) of X. We will where necessary introduce components of X that are not part of Motif. These other

components will always be treated as if they are to be used with Motif.

Other important concepts relating to more general Graphical User Interface (GUI) design and programming will be studied. In most cases this is directly in relation to the Motif programming model. However, Motif was designed to adhere to standard GUI design approaches and has guidelines defined in the *Motif Style Guide* (Chapter 20, [Ope93]).

## Conventions used

X provides functionality via a vast set of subroutine libraries. These may be called from a variety of high level languages.

They are most readily called from C programs as this is the language in which most of X is actually implemented in. We will only give C program examples in the main body of text. Appendix A gives details how to convert these examples into C++ and, more generally, write X/Motif applications in this language. Full program listings of both C and C++ are available from the online support and reference material [Mar96].

Readers should not be too worried about programming in C. We will not need to get heavily involved in C. Basically we will just be calling X/Motif functions and setting variables and data structures from our C programs. The ANSI C programming conventions are assumed in all examples.

In order to distinguish between program code and other text in this book the following fonts are used:

- All program code, fragments of code are highlighted in a typewriter type style.
- Motif and other X data types, structures, variables are also highlighted in a typewriter type style.
- Important definitions or concepts are highlighted in *italic* type.

We have already used the notation of X/Motif. For convenience throughout this book, a reference to X is taken to mean the X Window System. We refer to Motif whilst strictly speaking the full title is OSF/Motif where OSF stands for the Open Software Foundation the original developers of Motif.

## Graphical User Interfaces (GUIs)

Before we study Motif in detail it is worth considering why we need GUIs and how they can be effectively designed and used.

### Why Use GUIs?

GUIs provide an easy means of data entry and modification. They should provide an attractive and easy to use interface between human and machine. So easy in fact that a non-computer literate person could use the system.

GUI's provide a better means of communication than cumbersome text-based interfaces. Typically, GUIs provide such facilities by means of:

- Extensive use of visual control items -- Buttons, menus, icons, scroll bars *etc.*
- Intuitive on screen manipulation of data.
- If a standard GUI is adopted then consistency of use across platforms and applications is afforded. Nearly all MS Windows (or Apple Macintosh) have the same *look and feel* so the learning time for a new application is reduced.
- Multiple applications can be run simultaneously on most machines these days. GUIs provide better screen management of such processes -- we can assign one window (or more) to each application.

Although GUIs provide some very powerful advantages, there are a couple drawbacks to the GUI approach:

### Efficiency

-- As we will shortly see even the most basic windowing program can be quite large. This is because we will have to write or call upon many functions to control the windowing system -- *e.g.* create, move, resize *etc.* windows; handle input via mouse and keyboard actions; control graphics. Motif was designed as an attempt to reduce such problems by packaging common GUI entities together as *widgets*.

### Programming

-- A different approach to programming is needed from the traditional *command-line approach*. You have probably been used to the top down structured programming approach adhered to by languages such as Pascal and C. We will have to adopt a different strategy known as **event-driven** programming - where the actions in our program will be triggered by mouse and keyboard actions.

## Designing GUIs

The subject of Graphical User Interface design is large. Indeed it is a major topic in Computer Science in its own right. Consequently, we could devote the rest of the book to this topic. However many *standard* GUI design rules are prescribed by Motif. Many of these rules are prescribed automatically, whilst others are strongly suggested in the *Motif Style Guide* (Chapter 20,[Ope93]). In general it is the low level appearance and operation of specific objects (Buttons, Menus *etc.*) that are automatically facilitated by Motif. The higher level organisation of these objects is left to the control of the application developer. Some perhaps are fairly obvious, though not always strictly adhered to, rules of thumb for GUI design include:

- *Keep the interface as simple as possible* -- Do not over clutter a single interface.
- *Keep interfaces as consistent as possible* -- Adhere to standard GUI principles.
- *Keep in mind the application user* -- Provide easy access to common application interactions and do not over complicate common means of interaction.
- *Allow the user some control of the interface* -- This allows some customisation for user preferences.
- *Communicate the application actions to the user* -- Maintain a dialogue with the user. For example, do not allow the user believe that the system or application has "hung up" whilst performing intensive computations -- display a message or flash an icon to indicate some progress.

The *Motif Style Guide* (Chapter 20,[Ope93]) is a valuable source of information in relation to GUI design. Also the online reference material ([Mar96]) addresses further details on GUI design.

## History of X/Motif

This Section briefly surveys the important stages in the development of GUIs leading the X/Motif approach to GUI design and programming. Once X began to establish itself as one of the prime systems for window programming some issues still remained unresolved until recently, these are also briefly addressed.

## Communication before X

Computers first became commercially available in the 1950s. However, they were very large and expensive. They were also hard to program with very little thought was given to human computer interaction. By the 1960s very basic inroads into ease of computer use were being made with the development of more reliable operating systems. This was made possible as computers became smaller in size and more powerful in terms of processing ability. The first seeds of user interaction appeared in the form of early text editors during this period.

The start of the development of GUIs can be traced back to the early 1970's to Alan Kay's research group at Xerox's Palo Alto Research Centre. Two important projects were undertaken there:

- *Dynabook* (Early 1970s) -- where the goal was to produce a book sized personal computer with high resolution colour display and a radio link to a worldwide computer network. Mailbox, library, telephone and secretarial functions were also to be incorporated.
- *Star* (Late 1970s) where the goal was to produce a desk-sized personal workstation used by a single person. A high resolution display capable of fast high quality graphics was included. Graphical user interaction was provided by means of a mouse allowing options to be selected from a displayed menu. Later versions of *Star* introduced *icons* on the screen to represent objects and functions. The idea of *traits* -- a characteristic of an object that can be expressed by a set of methods or data and can be applied to, or carried by, the object holding that trait -- was also introduced. Traits were later to resurface in last release of Motif (2.0) (Chapter 21).

The first commercial exploitation of the Xerox work was realised in the early 1980s by Apple, firstly with *Lisa* and then the *Macintosh* series of computers. The Apple GUI proved very successful and popular and by the late 1980s many operating systems had adopted the GUI approach. UNIX vendors such as Sun (with SunView) and Dec (with DEC Windows) and Microsoft with Windows for the PC are examples.

There was one problem with the above developments. Every manufacturer had its own proprietary windowing system. These were all *entirely different* and different systems could not easily communicate with each other. It had been common to have networks of the same computers for some time and it was relatively easy to get machines of the same type in a network configuration to talk to each other.

The X Window system arose out of this very real need. The X system was designed to be platform independent and network-based. With X the programmer can write a single application in a single language and run this program on different machines with little or no modification. Moreover, applications can actually run programs on one computer and have the results displayed on another (or several) computer's terminal. The computer can be a similar model or an entirely different one altogether. The possibilities are endless.

## The Motif/Open Look War

Following the creation of the X Window system, two primary high level X interface toolkits came to

prominence:

### **Motif**

-- a product of the Open Software Foundation (OSF), an organization that originally included DEC, IBM and Hewlett-Packard.

### **Open Look/OpenWindows**

-- a product of Sun and AT&T.

Open Look was designed to support the X Window platform yet still maintain compliance with Sun's older native SunView Window system. Open Look had a slightly different design philosophy to Motif. Indeed for many years Sun claimed that Open Look was superior to Motif. At the time Sun were the substantial market leader vendor of UNIX platforms and therefore had a large influence on such matters. However, recently Sun decided to cease support of Open Look and adopt Motif.

Motif was based on IBM's Common User Access (CUA) guidelines as were both Microsoft Windows and OS/2. Consequently, the visual appearance and mode of operation, the so called *look and feel*, of Motif is similar to that of Microsoft Windows and OS/2. This was a deliberate strategy since there is sound business sense in profiting from an open system. More importantly, however, the predominance of Microsoft Windows in the PC market means that an interface that appears to the user to behave in a similar fashion to Microsoft Windows would be a logical choice in migrating (PC) applications to UNIX. It is probably a mixture of these factors that has led to Sun's decision to stop the development of Open Look and adopt Motif for Sun Workstations.

Following Sun's decision to support Motif, the Common Open Software Environment (COSE) united the major UNIX producers including Sun, DEC, IBM, Hewlett-Packard and UNIX System Laboratories. This has had a significant impact on the endorsement of Motif since it is now the standard choice for UNIX and general cross-platform GUI development. COSE has also prescribed choices of other X libraries concerned with 3D graphics (PEX) and Image extension (XIE) which are closely coupled to Motif.

## **Versions of Motif**

Motif has undergone four major revisions since its conception. Motif 1.0 is now quite old and should probably be avoided as there has been significant upgrades to Motif and the underlying X system in later Motif versions. Motif 1.1 was the next major release but this releases does not support some useful later features, such as drag and drop. However, many applications can still run under Motif 1.1. Motif 1.2 has been available since 1993 and is based on Release 5 of the Xlib and Xt specifications (X11R5). This version of Motif should be in common circulation now.

The latest version of Motif is version 2.0 and it was released in late 1994 as was the latest release of X11 (X11R6). Motif 2.0 provides some significant enhancements and many bug fixes. However, Motif 2.0 is not yet being shipped by the major UNIX vendors. The main reason is that most UNIX vendors made a decision to support a new *Common Desktop Envirnoment* (see Section 1.5.2 below) system, CDE 1.0, which uses Motif 1.2 and is not binary compatible with Motif 2.0. It is likely that these companies will not now deliver Motif 2.0 but wait to support the convergence of both products with the newer releases of Motif (2.1) and CDE (1.1). It is expected that both these will become available sometime in 1997.

Even though there have been four major revisions to Motif there have been several minor revisions to Motif. These were mainly fix bugs (sometimes a few hundred at a time !!). However different vendors do not always keep to consistent minor version release numbers.

The subset of Motif addressed in this book has remained more or less untouched by the developments in Motif 2.0. Where there are differences these are highlighted in the text. The major differences that concern us here are the support of additional widgets (Chapter 6 and C++ binding (Appendix A). Chapter 21 discusses the key features of Motif 2.0. The new features of Motif 2.0 are generally concerned with advanced uses of Motif, and are not within the defined scope of this text. All examples in this book have been tested extensively on Motif 1.2 and X11R5. There should be no problem in running these examples under Motif 2.0 or X11R6.

## Culture

In using and writing Motif programs you will inevitably be exposed to key parts of your computer. You will have to write Motif programs in a particular computer language -- usually C or C++. In writing and running Motif applications you will need to interact with the host operating system. You may also need to suitably equip your operating to run or display Motif applications. In conjunction with the operating system, there will be a windowing environment that controls how windows are displayed and managed in general. This section introduces these issues and explains how you configure your system to run Motif applications.

## Operating Systems

X Window is designed to be platform independent. Provided that machines which are connected to a network and are suitably equipped with software to run X Window, running applications across any kind of network is possible.

For Unix systems X/Motif is now the only practical window system available. Unix machines will usually be already configured to run some version of X. For users of PCs and Macintosh computers, the standard window environment is Microsoft Windows or the Macintosh Window Interface respectively. For users of these machines there are two options to set up X Window:

### X Window/Unix Systems

-- These basically convert you PC/Macintosh into a X Window/Unix operating system. Many allow the native PC/Mac operating or window system to coexist with the installed Unix system. Many commercial and freeware packages exist for running Unix and the basic X system. Motif libraries have to be purchased to run on top of the basic X system. Probably the most popular system is the freely available Linux system. Linux is available for both PC and Macintosh (Power PC Macs only). Linux basically converts the host machine into a Unix environment. X/Motif libraries are available for Linux at a small additional cost. For further details on Linux consult the following URLs:

<http://www.linux.org/>

-- Main Linux site home page.

<http://www.linux.co.uk/>

-- U.K. Linux Site.

<http://www.rahul.net/kenton/xsites.html>

-- General X Window and Linux information with many links to related sites

<http://www.mklinux.apple.com/>

-- Linux on the Macintosh.

There are a few other commercial packages available that run Unix/X Window environments on a PC and Macintosh.

## Window Display/Server Software

-- Some commercial packages are available that allow the host machine to act as an X server and/or an X display. Some packages act as an X display meaning that X application must be run on a machine that supports an X server but the (X) output of the application can be redirected to X display.

Packages for the PC that allow this include SunSoft Inc.'s SolarNet PC-X 1.1, Hummingbird Communications Ltd. *eXceed 5 for Windows*, Network Computing Devices Inc.'s *PC-Xware 3.0* and Walker Richer and Quinn Inc.'s *Reflection Suite for Windows 5.0*.

Package available to allow a Macintosh to become an X Server include Apple's *MacX*, Intercon's *Planet X*, Netmanage/AGE's *XoftWare*, Tenon *XTen* and White Pine's *eXodus*.

A free X server for PCs and Macintosh exists called *MI/X*.

ie WWW site URL:<http://www.rahul.net/kenton/xsites.html> contains links to almost all the ove systems.

ote that minimum configurations of many computers in terms of processor speed and/or memory are likely to be adequate to support X Window.

## The Common Desktop Environment (CDE)

uilt on top of the operating system is the windowing environment. This environment controls how indows are displayed and how events invoked by mouse selection, keystrokes *etc.* are processed. The neral term for the housekeeping of windows is *window management* and further details on this will be scussed in Section 3.2. X window toolkits, such as Motif and Open Look, generally provide two key mponents: the window manager and the toolkit libraries.

ie window manager basically defines the *look and feel* of a particular toolkit. However, with ever creasing windowing needs, the basic window manager and related system and common application needs ve grown into a *desktop environment* providing a whole suite of tools including a window manager . The ification of the Unix and X Window providers with COSE has led to the vendors developing a unified sktop for Unix systems -- the *Common Desktop Environment* (CDE) [Add94g,Add94a,Add94e,Add94f,Add94c,Add94d,Add94b]. The CDE is built on top of Motif.

ie CDE was designed to provide end users with a consistent graphical user interface across workstations d PCs, and software developers with a single set of programming interfaces for platforms that support the Window System and Motif.

ie CDE is intended to:

- Reduce learning time by providing the same appearance and behaviour across multiple operating systems.
- Increase productivity by helping system administrators and end users customize the desktop environment to fit individual work styles and preferences.
- Make learning easier by providing a consistent, rich, and easily accessible context-sensitive on-line help system for help whenever and wherever the user needs it.
- Provide a common set of desktop and application development tools.

Throughout the development of X/Motif some key companies and consortia have been responsible for key aspects of the system. We briefly summarise these contributions in this Section. Another key aspect to the development of Motif is the prescribed *uniformity* of Motif applications. The *Motif Style Guide* is the main reference to Motif application development.

## OSF, X Consortium, Open Group

The Open Software Foundation consortium provides many services and is not solely concerned with X Window related matters. The OSF licenses Motif, offers training courses, testing and certification of software intended for commercial use.

The X Consortium distributes the X System and manuals at a minimal cost (basically the cost of distribution media and shipping). Motif is built on top of the X System.

The Open Group was formed in February, 1996 by the consolidation of the two leading open systems consortia, X/Open Company Ltd. and the Open Software Foundation (OSF). Under the Open Group umbrella, OSF and X/Open work together to deliver technology innovations and wide-scale adoption of open systems specifications. From the beginning of 1997 the Open Group will have responsibility for the X Window System transferred from the X Consortium.

For further information, the OSF, X Consortium and the Open Group can be contacted at the following addresses:

**Open Software Foundation, 11 Cambridge Center Cambridge MA 02142. Tel (USA): 617/621-8700.**  
**Email: [info@osf.org](mailto:info@osf.org) WWW:**  
**<http://www.osf.org/>**

**X Consortium Inc., One Memorial Drive PO BOX 546 Cambridge MA 02142-0004. Tel**  
**(USA): 617/374-1000. WWW:**  
**<http://www.x.org/>**

**Open Group, 11 Cambridge Center, Cambridge, MA 02142: Tel (USA): 617/621-7300. Email:**  
**[info@opengroup.org](mailto:info@opengroup.org) WWW:**  
**<http://www.opengroup.org/>**

## Motif and COSE

The Common Open Software Environment (COSE) was formed when the major UNIX producers, including Sun, DEC, IBM, Hewlett-Packard and UNIX system Laboratories, decided to unite and attempt to standardise UNIX implementations worldwide in 1993. The remit of COSE is to define standard cross-platform UNIX systems incorporating application program interfaces, windows interfacing, desktop environments, graphics, multimedia, system management, support for distributed computing and large scale data management. The standardisation of a Common Desktop Environment (CDE) for UNIX resulted in the adoption of Motif for this purpose.

As such, the OSF, Open Group, and COSE can be regarded as the *elders* of Motif and future releases of Motif will be determined by them.

In late 1995, The OSF announced the formal signing of the Joint Development agreement for the further enhancement and evolution of the Common Desktop Environment and OSF/Motif under the OSF



Prestructured Technology (PST) development process.

## Motif Style Guide

The *Motif Style Guide* [Ope93] can be regarded as the *Bible* for Motif Application developers. Along with a Motif Reference Manual [Mar96,Hel94b] and a programming text book (hopefully this text), the *Motif Style Guide* provides an invaluable source of information.

The *Motif Style Guide* provides a set of guidelines that provides a framework for the behaviour of Motif application developers, GUI developers, widget developers and window managers. The basic idea is that all Motif applications that adhere to the prescribed style will maintain a high level of consistency. Also, since Motif follows CUA guidelines, Motif applications will be similar to Microsoft Windows applications in terms of appearance and user operation. For developers of commercial applications, the adoption of Motif style is critical.

Many standard GUI issues are integrated into a Motif Widget default behaviour. Therefore, these defaults should only be modified with great care and consideration for such implications. Other aspects of style are left to the developer. The *Motif Style Guide* only *suggests* certain standard operations.

As has been mentioned, the *Motif Style Guide* has a greater scope than just the Motif application developer (the intended audience of this text). Chapter 20 summarises many important issues relating to the application developer. Where appropriate, specific style considerations are mentioned elsewhere in this text.

## Getting There

Throughout the long journey to learn Motif there are many useful sources of reference that could ease the burden. In recent years the Internet and the World Wide Web (WWW) has established itself as a very useful source for a variety of information types. X/Motif is no exception and the major distributors of X/Motif have WWW, email and ftp address. Many sites also contain frequently asked questions about general and specific X/Motif queries. News updates on major and minor release of X/Motif and tutorial material is also plentiful. This Chapter provides pointers to large repositories of such information in a variety of Internet and WWW distribution mediums.

## Motif distribution

Various components of X/Motif are distributed by the OSF, the X Consortium, the Open Group and by a number of independent vendors for a variety of platforms. Section 1.6.1 gives details on these matters.

## WWW and Ftp Access

The main WWW source of information for motif is *MW3: Motif on the World Wide Web* (URL: <http://www.cen.com/mw3/>). From the home page you can connect to a wealth of resources for Motif and X Window System development. MW3 presently contains over 700 links. Virtually all aspects of Motif are

covered here.

Other useful WWW links include:

- <http://www.rahul.net/kenton/xsites> -- Good source of X information and links to many related sites.
- <http://www.landfield.com/faqs/faqsearch.html> -- The best frequently asked questions (FAQ) search interface (Usenet Hypertext FAQ Archive).
- X, Xt and Motif FAQ are also archived at:
  - Utrecht University ( <http://www.cs.ruu.nl/wais/html/na-dir/>),
  - Oxford University ( <http://www.lib.ox.ac.uk/internet/news/faq/>),
  - SUNSite Northern Europe ( <http://src.doc.ic.ac.uk/usenet/usenet-by-hierarchy/comp/windows/x/>).

The Motif FAQ is available via ftp also at:

- Century Computing Inc, USA -- The file is available in raw text and compressed formats:  
<ftp://ftp.cen.com/pub/Motif-FAQ>,  
<ftp://ftp.cen.com/pub/Motif-FAQ.Z> and <ftp://ftp.cen.com/pub/Motif-FAQ.gz>.
- MIT -- The Motif FAQ is available in 9 parts: <ftp://rtfm.mit.edu/pub/usenet-by-group/comp.windows.x.motif>.
- X Consortium -- <ftp://ftp.x.org/contrib/faqs/Motif-FAQ>.

## Things to take

Other sources of information on the Internet are provided via *mailing lists* and *news groups*. Mailing lists are sent via email and serve as discussion groups and avenues for news announcements for particular topics. News groups can be read by specific *news reader* applications and broadly serve as discussion groups. Mailing lists and news groups do not necessarily require a WWW browser for reading although browsers such as Netscape Navigator do provide specific access to news groups and email.

## Mailing lists

The following public mailing lists are maintained by the X Consortium for the discussion of issues relating to the X Window System. All are hosted @x.org.

### xpert

A mailing list that discusses many X related issues. This list is gatewayed to the newsgroup comp.windows.x (*see below*). To subscribe to this mailing list, send mail to the request address. In general this is specified by adding -request to the name of the desired list. Thus, to *add* yourself to the *xpert* mailing list:

```
To: xpert-request@x.org
Subject: (none needed)
```

```
subscribe
```

To *unsubscribe*:

```
To: xpert-request@x.org
Subject: (none needed)
```

unsubscribe

To add an address to a specific list, or to add a specific user, you can specify options to the subscribe or unsubscribe command. This example adds dave@widget.uk to the xpert mailing list:

```
To: xpert-request@x.org
Subject: (none needed)

subscribe xpert dave@widget.uk
```

### xannounce

This is a moderated mailing list for announcing releases of non-commercial X related software, and other issues concerning the X Window System.

This mailing list is gatewayed to the newsgroup *comp.windows.x.announce*.

Subscription requests should be sent to *xannounce-request@x.org*.

## News groups

The news group **comp.windows.x.motif** is the main news group for Motif related issues. The following news groups exist for the discussion of other issues related to the X Window System:

### comp.windows.x

-- This news group is gatewayed to the xpert mailing list (*see above*).

### comp.windows.x.announce

This group is moderated by the staff of X Consortium, Inc. Traffic is limited to major X announcements, such as X Consortium standards, new releases, patch releases, toolkit releases and X conferences, exhibitions, or meetings.

### comp.windows.x.apps

-- This news group is concerned with X applications.

### comp.windows.x.intrinsics

-- This news group is concerned with Xt toolkit.

### comp.windows.x.pex

-- This news group is concerned with the 3D graphics extension to X.

### alt.windows.cde

-- The news group dedicated to Common Desktop Environment issues.

Most of the above news groups have a frequently asked question section posted regularly to the news group which provide valuable information for the novice and discuss common problems. The *comp.windows.x.motif* are also accessible from many of the WWW sites listed in Section [21.3.2](#)

## Environment -- The X Window System

## What is the X Window system?

The *X Window* system provides a way of writing device independent graphical and windowing software that can be easily ported from machine to machine. X is a network-based system and supports cross-platform communication -- we can get different machines to talk to each other.

At the highest level of X there are two basic features: The *window manager* and the *toolkit*. The window manager (wm) controls GUI aspects such as appearance of windows and interaction with the user. The toolkits are C subroutine libraries where we describe how to construct the GUI and how to attach the GUI to the remainder of the application. Motif is one such toolkit. Motif is built on other toolkits in the X System: Xt Intrinsics, an intermediate level toolkit, and the low level X Library (Xlib). The relevance of these will be made apparent in due course.

## X Window Principles

All forms of displaying of information in X are *bit-mapped* which means that every pixel on the screen is individually controllable. Therefore we can draw pictures and use text. The requirement for bit-mapped graphics means that high quality monitors are necessary. However, most computer systems provide monitors of this type.

X, like most other windowing systems, divides the screen into various parts that control input and output. Each part is called a *window*. A window can have many uses. A window can display graphics, receive input from a mouse, act as a standard terminal (e.g. an Xterm -- a standard text based terminal emulation window) etc..

Not all applications need to consist of a single window. We can have many windows associated with different parts of one application. Each subwindow is called a *child* and it usually remains under the control of its *parent* window.

There is one special window, the background or *root* window. All other windows are children of the root.

## The Window Manager

The *window manager* is responsible for manipulating windows on the screen. The window manager performs the following operations:

- Placement and movement of windows.
- Resizing of windows.
- Iconification of windows -- how the window appears when the window is *minimised*.
- Starting and manipulation of windows.
- Control of input to windows.

Controlling the window environment is not easy and has many facets. For instance, there may be multiple applications running simultaneously and a conflict may arise for input:

*Does a keyboard input go in a window where the mouse currently points or must a window be explicitly chosen?*


The window manager is also predominantly responsible for the appearance and user interaction (the *look and feel*) of the interface. Since the development of X, there have been a few different window managers. The look and feel varies a lot between each window manager. The Motif window manager (*mwm*) is probably the window manager you are most familiar with as this comes with the Motif system. Other window managers include Sun's Open Look window manager (*olwm*) and the Tab or Tom's window manager (*twm*) .

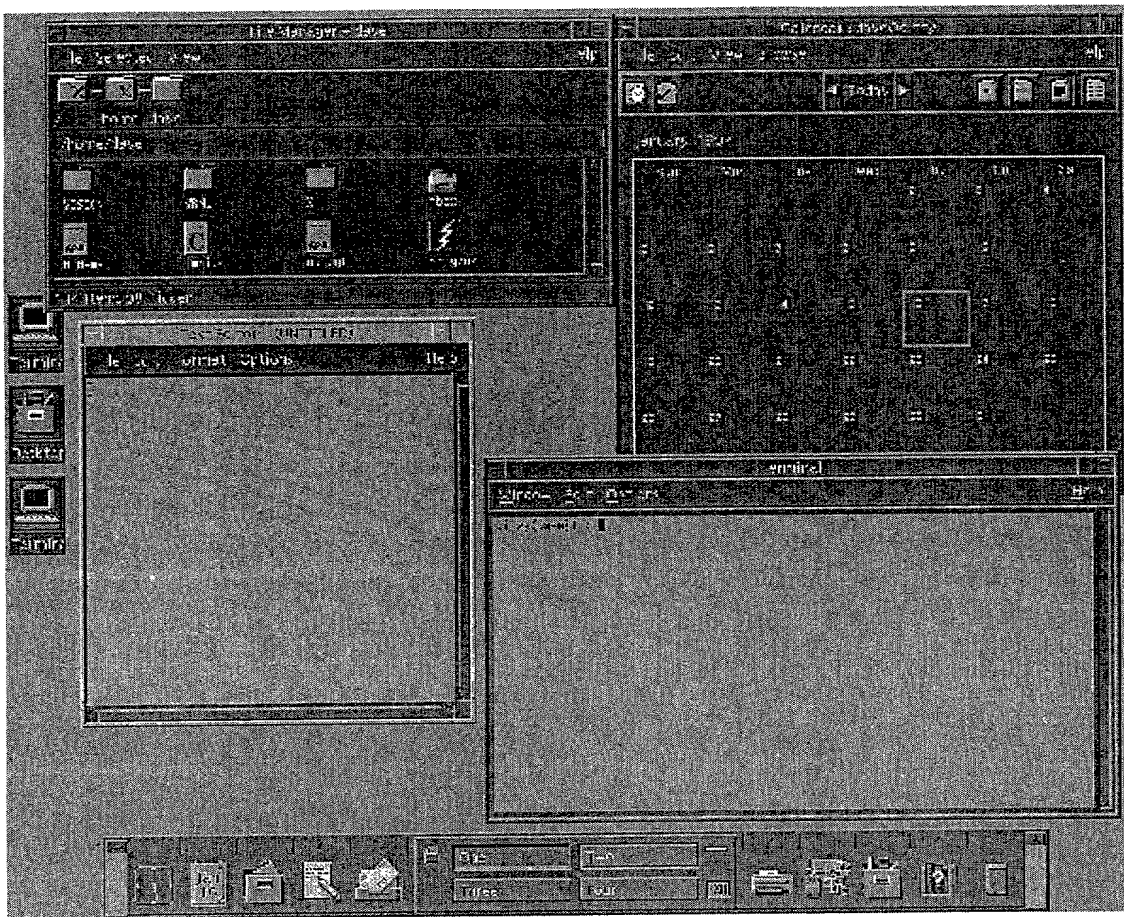
Most major Unix vendors now supply the CDE which by default runs the desktop window manager, *dtwm* . The development of the common desktop will probably result in *dtwm* superseding *mwm*. The CDE default window manager can easily be altered to run the above alternative window managers if desired. Since the CDE is built on top of motif there are many similarities between *dtwm* and *mwm* (see Section 3.4 below).

The Motif *look and feel* , as defined by the *Motif Style Guide*, is basically enforced by *mwm* or *dtwm*. Consequently, interaction between applications and these window managers are eased if the applications obey standard Motif/CDE guidelines.

## The Common Desktop Environment

Most major Unix vendors now provide the CDE as standard. Consequently, most users of the X Window system will now be exposed to the CDE. Indeed, continuing trends in the development of Motif and CDE will probably lead to a convergence of these technologies in the near future. This section highlights the key features of the CDE from a Users perspective.

Upon login, the user is presented with the CDE Desktop (Fig. ). The desktop includes a front panel (Fig. 3.2) , multiple virtual workspaces, and window management. CDE supports the running of applications from a file manager, from an application manager and from the front panel. Each of the subcomponents of the desktop are described below.



**Fig. 3.1 Sample CDE Desktop**

## The front panel

The front panel (Fig. 3.2) contains a set of icons and popup menus (more like roll-up menus) that appear at the bottom of the screen, by default (Fig. 3.1). The front panel contains the most regularly used applications and tools for managing the workspace. Users can drag-and-drop application icons from the file manager or application manager to the popups for addition of the application(s) to the associated menu. The user can also manipulate the default actions and icons for the popups. The front panel can be locked so that users can't change it. A user can configure several virtual workspaces -- each with different backgrounds and colors if desired. Each workspace can have any number of applications running in it. An application can be set to appear in one, more than one, or all workspaces simultaneously.

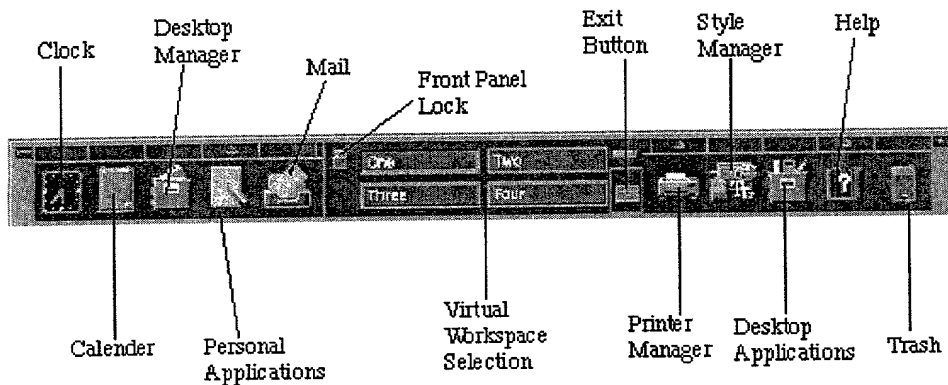


Fig. 3.2 CDE Front Panel

## The file manager

CDE includes a standard file manager. The functionality is similar to that of the Microsoft Windows, Macintosh, or Sun Open Look file manager. Users can directly manipulate icons associated with UNIX files, drag-and-drop them, and launch associated applications.

## The application manager

The user interaction with the application manager is similar to the file manager except that it is intended to be a list of executable modules available to a particular user. The user launches the application manager from an icon in the front panel. Users are notified when a new application is available on a server by additions (or deletions) to the list of icons in the application manager window. Programs and icons can be installed and pushed out to other workstations as an integral part of the installation process. The list of workstations that new software is installed on is configurable. The application manager comes preconfigured to include several utilities and programs.

## The session manager

The session manager is responsible for the start up and shut down of a user session. In the CDE, applications that are made *CDE aware* are warned via an X Event when the X session is closing down. The application responds by returning a string that can be used by the session manager at the user's next login to restart the application. CDE can remember two sessions per user. One is the *current* session, where a snapshot of the currently running applications is saved. These applications can be automatically restarted at the user's next login. The other is the default login, which is analogous to starting an X session in the Motif window manager. The user can choose which of the two sessions to use at the next login.

## Other CDE desktop tools

CDE 1.0 includes a set of applications that enable users to become productive immediately. Many of these are available directly from the front panel, others from the desktop or personal application managers. Common and productive desktop tools include:

### **Mail Tool**

-- Used to compose, view, and manage electronic mail through a GUI. Allows the inclusion of attachments and communications with other applications through the messaging system.

### **Calendar Manager**

-- Used to manage, schedule, and view appointments, create calendars, and interact with the Mail Tool.

### **Editor**

-- A text editor with common functionality including data transfer with other applications via the clipboard, drag and drop, and primary and quick transfer.

### **Terminal Emulator**

-- An *xterm* terminal emulator.

### **Calculator**

-- A standard calculator with scientific, financial, and logical modes.

### **Print Manager**

-- A graphical print job manager for the scheduling and management of print jobs on any available printer.

### **Help System**

-- A context-sensitive graphical help system based on Standard Generalized Markup Language (SGML).

### **Style Manager**

-- A graphical interface that allows a user to interactively set their preferences, such as colors, backdrops, and fonts, for a session.

### **Icon Editor**

-- This application is a fairly full featured graphical icon (pixmap) editor.

## **Application development tools**

CDE[[Add94g](#),[Add94a](#),[Add94e](#),[Add94f](#),[Add94c](#),[Add94d](#),[Add94b](#)] includes two components for application development. The first is a shell command language interpreter that has built-in commands for most X Window system and CDE functions. The interpreter is based on ksh93[[Add93b](#),[Add93a](#)], and should provide anyone familiar with shell scripts the ability to develop X, Motif, and CDE applications.

To support interactive user interface development, developers can use the Motif Application Builder. This is a GUI front end for building Motif applications that generates C source code. The source code is then compiled and linked with the X and Motif libraries to produce the executable binary.

## **Application integration**

CDE provides a number of tools to ease integration. The overall model of the CDE session is intended to allow a straightforward integration for virtually all types of applications. Motif and other X toolkit applications usually require little integration.



The task of integrating in-house and third party applications into a desktop, often the most difficult aspect of a desktop installation, is simplified by CDE. The power and advantage of CDE functionality can be realized in most cases without recompiling applications.

For example, Open Look applications can be integrated through the use of scripts that perform front-end execution of the application and scripts that perform pre- and post-session processing.

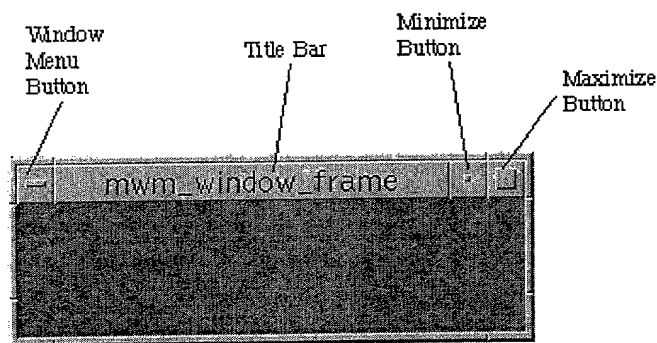
After the initial task of integrating applications so that they fit within session management, further integration can be done to increase their overall common *look-and-feel* with the rest of the desktop and to take advantage of the full range of CDE functionality. Tools that ease this aspect of integration include an *Icon Editor* used to create colour and monochrome icons. Images can be copied from the desktop into an icon, or they can be drawn freehand.

The *Action Creation Utility* is used to create action entries in the action database. Actions allow applications to be launched using desktop icons, and they ease administration by removing an application's specific details from the user interface.

The *Application Gather* and *Application Integrate* routines are used to control and format the application manager. They simplify installations so that applications can be accessible from virtually anywhere on the network.

## Windows and the Window Manager

From a user's perspective, one of the first distinguishing features of Motif's *look and feel* is the *window frame* (Fig. 3.3). Every application window is contained inside such a frame. The following items appear in the window frame:




**Fig. 3.3 The Motif Window Frame**

### Title Bar

-- This identifies the window by a text string. The string is usually the name of the application program. However, an application's resource controls the label (Chapter [□](#)).

### Window Menu

-- Every window under the control of *mwm* has a window menu. The application has a certain amount of control over items that can be placed in the menu. The *Motif Style Guide* insists that certain commands are always available in this menu and that they can be accessed from either mouse or keyboard selection. Keyboard selections are called *mnemonics* and allow routine actions (that may

involve several mouse actions) to be called from the keyboard. The action from the keyboard usually involves pressing two keys at the same time: the *Meta* key  and another key. The default window menu items and *mnemonics* are listed below and illustrated in Fig. 3.4:

Resto <u>r</u> e	Alt+F5
M <u>o</u> ve	Alt+F7
M <u>i</u> nimize	Alt+F9
Ma <u>x</u> imize	Alt+F10
Lo <u>w</u> er	Alt+F3
<hr/>	
O <u>cc</u> upy Workspace...	
O <u>cc</u> upy A <u>ll</u> Workspaces	
Un <u>o</u> ccupy Workspace	
<hr/>	
C <u>l</u> ose	Alt+F4

Fig. 3.4 The Window Menu

- **Restore (Meta+F5)** -- Restore window to previous size after iconification (*see* below).
- **Move (Meta+F7)** -- Allows the window to be repositioned with a drag of the mouse.
- **Size (Meta+F8)** -- Allows the size of the window to be changed by dragging on the corners of the window.
- **Minimize (Meta+F9)** -- Iconify the window.
- **Maximize (Meta+F10)** -- Make the window the size of the root window, usually the whole of the display size.
- **Lower (Meta+F3)** -- Move the window to the bottom of the window stack. Windows may be *tiled* on top of each other (*see* below). The front window being the top of the stack.
- **Close (Meta+F4)** -- Quit the program. Some simple applications (Chapter 5) provide no *internal* means of termination. The `Close` option being the only means to achieve this.

**Minimize Button**

-- another way to iconify a window .

**Maximize Button**

-- another way to make a window the size of the root window .

The window manager must also be able to manage multiple windows from multiple client applications. There are a few important issues that need to be resolved. When running several applications together, several windows may be displayed on the screen. As a result, the display may appear cluttered and hard to navigate. The window manager provides two mechanisms to help deal with such problems:

**Active Window**

-- Only one window can receive input at any time. If you are selecting a graphical object with a mouse, then it is relatively easy for the window manager to detect this and schedule appropriate actions related to the chosen object. It is not so easy when you enter data or make selections directly from the keyboard. To resolve this only one window at a time is allowed *keyboard focus*. This window is called the *active window*. The selection of the active window will depend on the system configuration which the user typically has control over. There are two common methods for selecting the active window:

**Focus follows pointer**

-- The active window is the window is the window underneath mouse pointer.

**Click-to-type**

-- The active window is selected, by clicking on an area of the window, and remains active until another window is selected no matter where the mouse points.

When a window is made active its appearance will change slightly:

- Its outline frame will become shaded.
- The cursor will change appearance when placed in the window.
- The window may jump, or be *raised* to the top of the window stack.

The exact appearance of the above may vary from system to system and may be controlled by the user by setting environment settings in the window manager.

## Window tiling

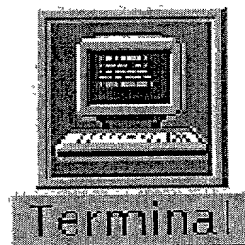
-- Windows may be stacked on top of each other. The window manager tries to maintain a three-dimensional *look and feel*. Apart from the fact that buttons, dialog boxes appear to be elevated from the screen, windows are shaded and framed in a three-dimensional fashion. The top window (or currently active window) will have slightly different appearance for instance.

The window menu has a few options for controlling the tiling of a window. Also a window can be brought to the top of the stack, or *raised* by clicking a part of its frame.

## Iconification

-- If a window is currently active and not required for input or displaying output then it may be *iconified* or *minimised* thus reducing the screen clutter. An icon (Fig. 3.5) is a small graphical symbol that represents the window (or application). It occupies a significantly less amount of screen area. Icons are usually arranged around the perimeter (typically bottom or left side) of the screen. The application will still be running and occupying computer memory. The window related to the icon may be reverted to by either double clicking on the icon, or selecting *Restore* or *Maximise* from the icon's window menu.

**Figure 3.5:** Sample  
Icon from Xterm  
Application



# The Root Menu

The *Root Menu* is the main menu of the window manager. The root menu typically is used to control the whole display, for example starting up new windows and quitting the desktop. To display the Root menu:

- Move the mouse pointer to the Root Window.
- Hold down the left mouse button.

The default Root Menu has the following The root menu can be customised to start up common applications for example. The root menu for the *mwm* (Fig. 3.6) and *dtwm* (Fig. 3.7) have slightly different appearance but have broadly similar actions, which are summarised below:

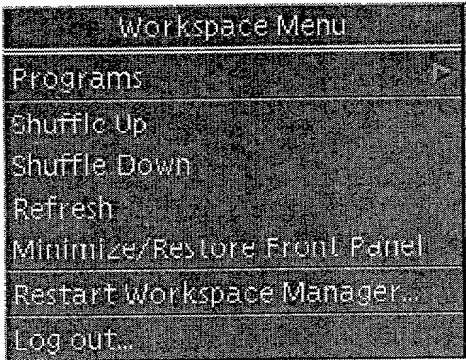
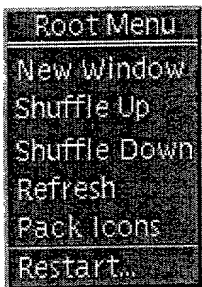


Fig. 3.6 The *mwm* Root Menu

Fig. 3.7 The CDE *dtwm* Root Menu

**Program**

(*dtwm*) -- A sub-menu is displayed that allows a variety of programs to be called from the desktop, for example to create a new window. The list of available programs can be customised from the desktop.

**New Window**

(*mwm*) -- Create a new window which is usually an *Xterm* window.

**Shuffle Up**

-- Move the bottom of the window stack to the top.

**Shuffle Down**

-- Move the top of the window stack to the bottom.

**Refresh**

-- Refresh the current screen display.

**Restart**

-- Restart the Workspace.

**Logout**

(*dtwm*) -- Quit the Window Manager.

# Exercises

**Exercise 8179**

Compare OPEN LOOK and MOTIF window managers CDE ON SUN???.

## Exercise 8180

Add applications to the application manager

# The X Window Programming Model

This Chapter introduces the basic concepts and principles that are of concern to the Motif programmer. We define basic system concepts, describe the three basic levels of the X programming model and describe basic Motif components.

## X System Concepts and Definitions

X requires a system that consists of workstations capable of bit-mapped graphics. These can be colour or monochrome.

A *display* is defined as a workstation consisting of a keyboard, a pointing device (usually a mouse although it could be a track ball or graphics tablet, for instance) and one or more screens.

## Clients and Servers

X is network oriented and applications need not be running on the same system as the one supporting the display. This can sometimes be quite complicated for a system such as X to manage and so the concept of *clients* and *servers* was introduced.

You need not worry too much about the practicality of this, as normally X makes this transparent to the user -- especially if we run programs on a single workstation. However, in order to fully understand the workings of X, some notion of these concepts is required.

The program that controls each display is known as the *server*. This terminology may seem a little odd as we may be used to the server as something across the network such as a file server. *Here*, the server is a local program that controls our *display*. Also our display may be available to other systems across the network. In this case our system does act as a true display server.

The server acts as a go-between between user programs, called *clients* or applications and the resources of the local system. These run on either local or remote systems.

Tasks the server performs include:

- allowing access by multiple clients,
- interpreting network messages from clients,
- two-dimensional graphics display,
- maintain local resources such as windows, cursors, fonts and graphics.

# The X Programming Model

The client and server are connected by a communication path called ( *surprise, surprise*) the *connector* . This is performed by a low-level C language interface known as *Xlib* . Xlib is the lowest level of the X system software hierarchy or architecture (Fig 4.1). Many applications can be written using Xlib alone. However, in general, it will be difficult and time consuming to write complex GUI programs only in Xlib. Many *higher level* subroutine libraries, called **toolkits** , have been developed to remedy this problem.

**Note:** X is not restricted to a single language, operating system or user interface. It is relatively straightforward to link calls to X from most programming languages. An X application must only be able to generate and receive messages in a special form, called *X protocol messages* . However, the protocol messages are easily accessible as C libraries in Xlib (and others).

There are usually two levels of toolkits above Xlib

- **X Toolkit (Xt) Intrinsics** are parts of the toolkit that allow programmers to build new widgets.
- **Third Party Toolkits** -- such a Motif .

An application program in X will usually consist of two parts. The graphical user interface written in one or more of Xlib, Xt or Motif and the algorithmic or functional part of the application where the input from the interface and other processing tasks are defined. Fig. 4.1 illustrates the relationships between the application program and the various parts of the X System.



**Fig. 4.1 The X Programming Model** The main concern of this text is to introduce concepts in building the graphical user interface in X and Motif in particular. We now briefly describe the main tasks of the three levels of the X programming model before embarking on writing Motif programs.

## Xlib

The main task of Xlib is to translate C data structures and procedures into the special form of X protocol messages which are then sent off. Obviously the converse of receiving messages and converting them to C structures is performed as well. Xlib handles the interface between client (application) and the network.

## Xt Intrinsics

Toolkits implement a set of user interface features or application environments such as menus, buttons or scroll bars (referred to as **widgets**).

They allow applications to manipulate these features using object-oriented techniques.

*X Toolkit Intrinsics* or *Xt Intrinsics* are a toolkit that allow programmers to create and use new widgets.

If we use widgets properly, it will simplify the X programming process and also help preserve the *look and feel* of the application which should make it easier to use.

We will have to call some Xt functions when writing Motif programs since Motif is built upon Xt and thus needs to use Xt. However, we do not need to fully understand the workings of Xt as Motif takes care of most things for us.

## The Motif Toolkit

X allows extensions to the *Xt Intrinsics* toolkit. Many software houses have developed custom features that make the GUI's appearance attractive, easy to use and easy to develop. *Motif* is one such toolkit.

The third party toolkits usually supply a special client called the **window manager**

## Currency

The basic unit of currency of Motif is the *widget*. The widget is the basic building block for the GUI. It is common and beneficial for most GUIs assembled in Motif to look and behave in a similar fashion. Motif enforces many of these features by providing default actions for each widget. Motif also prescribes certain other actions that should, whenever possible, be adhered to. Information regarding Motif GUI design is provided in the *Motif Style Guide* [Ope93]. We now briefly address general issues relating to Motif widgets and style.

## Widget Classes and Hierarchies

A *widget*, in Motif, may be regarded as a general abstraction for user-interface components. Motif provides widgets for almost every common GUI component, including buttons, menus and scroll bars. Motif also provides widgets whose only function is to control the layout of other widgets -- thus enabling fairly advanced GUIs to be easily designed and assembled.

A widget is designed to operate independently of the application except through well defined interactions, called *callback functions*. This takes a lot of mundane GUI control and maintenance away from the application programmer. Widgets know how to redraw and highlight themselves, how to respond to certain events such as a mouse click *etc.* Some widgets go further than this, for example the Text widget is a fully functional text editor that has built in cut and paste as well as other common text editing facilities.

The general behaviour of each widget is defined as part of the Motif (Xm) library. In fact Xt defines certain base classes of widgets which form a common foundation for nearly all Xt based widget sets. Motif provides a *widget set*, the Xm library, which defines a complete set of widget classes for most GUI requirements on top of Xt (Fig 4.1).

The Motif Reference Manual [Hel94b] provides definitions on all aspects of widget behaviour and interaction. Basically, each widget is defined as a C data structure whose elements define a widget's data attributes, or *resources* and pointers to functions, such as *callbacks*.

Each widget is defined to be of a certain *class*. All widgets of that class inherit the same set of resources and callback functions. Motif also defines a whole hierarchy of widget classes. There are two broad Motif widget classes that concern us. The *Primitive* widget class contains actual GUI components, such as buttons and text widgets. The *Manager* widget class defines widgets that hold other widgets.

Chapter 5 introduces basic Motif widget programming concepts and introduces how resources and callback functions are set up. Chapter 6 then goes on to fully define each widget class and the Motif widget class hierarchy. Following Chapters then address each widget class in detail.

## Motif Style -- GUI Design

The *Motif Style Guide* should be read by every Motif Application developer. The Style Guide is not intended to be a Motif programming manual. This book is not intended to be a complete guide to Motif style. The books should be regarded as essential companions along with a good Motif reference source.

The *Motif Style Guide* provides a set of guidelines that specify a framework for the behaviour of Motif application developers, GUI developers, widget developers and window managers. Many standard GUI design and behaviour issues are integrated into a Motif widget's default settings. Therefore these defaults should only be modified with great care and consideration. Other aspects of style are left to the developer. The *Motif Style Guide* only suggests certain standard operations, but where appropriate these should be adopted.

The *Motif Style Guide* prescribes many common forms of interaction and interface design. For example, it defines how menus should be constructed, used and organised. Chapter 20 summarises all the common style concerns for the Motif programmer. Where appropriate specific reference is made in individual Sections to Motif style for a particular widget.

## A First Motif Program

In this Chapter we will develop our first Motif program. The main purpose of the program is to illustrate and explain many of the *fundamental* steps of nearly every Motif program.

## What will our program do?

Our program, `push.c`, will create a window with a single push button in it. The button contains the string, "Push Me". When the button is pressed (with the *left* mouse button) a string is printed to *standard output*. We are not yet in a position to write back to any window that we have created. Later Chapters will explore this possibility. However, this program does illustrate a simple interface between Motif GUI and the application code.

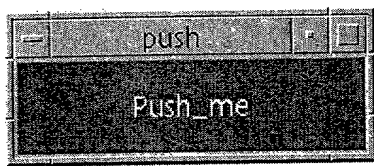
The program also runs forever. This is a key feature of event driven processing. For now we will have to quit our programs by either:

- Using the Operating System to terminate the program (process) -- there are many ways in which this could be done. The easiest way is to use `ctrl-c` to quit from the command line.
- Use the *Window Menu* quit option (Section 3.4) -- depress *right mouse* down around the top perimeter of the window and choose the *quit* option from menu. The *OSF/Motif Style Guide* (Chapter 20, [Ope93]), in common with standard GUI practice, also prescribes that *hot keys*, or *keyboard shortcuts*, or *mnemonics* should be facilitated so that common actions can be performed from the keyboard. It is standard convention that the **Meta-F4** is used to close an application.



In forthcoming Chapters (see also Exercise [5.1](#)) we will see how to quit the program from within our programs.

The display of `push.c` on screen will look like this:



**Fig. 5.1** Push.c display

## What will we learn from this program?

As was previously stated, the main purpose of studying the program is to gain a fundamental understanding of Motif programming. Specifically the lessons that should be clear before embarking on further Motif programs are:

- How to write and compile a Motif Program .
- The relationship between Xlib, Xt Intrinsics and Motif .
- How to create simple widgets and manage its resources .
- How widgets handle events .
- How to call functions from events -- the Interface between the Motif GUI and the application code .

We now list the complete program code and then go on to study the code in detail in the remainder of this Chapter.

## The `push.c` program

The complete program listing for the `push.c` program is as follows:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

/* Prototype Callback function */

void pushed_fn(Widget , XtPointer ,
               XmPushButtonCallbackStruct *);

main(int argc, char **argv)
{
    Widget top_wid, button;
    XtAppContext app;

    top_wid = XtVaAppInitialize(&app, "Push", NULL, 0,
                               &argc, argv, NULL, NULL);

    button = XmCreatePushButton(top_wid, "Push_me", NULL, 0);
```

```

/* tell Xt to manage button */
XtManageChild(button);

/* attach fn to widget */
XtAddCallback(button, XmNactivateCallback, pushed_fn, NULL);

XtRealizeWidget(top_wid); /* display widget hierarchy */
XtAppMainLoop(app); /* enter processing loop */
}

void pushed_fn(Widget w, XtPointer client_data,
               XmPushButtonCallbackStruct *cbs)
{
    printf("Don't Push Me!!\n");
}

```

## Calling Motif, Xt and Xlib functions

When writing a Motif program you will invariably call upon **both** Motif and Xt functions and data structures *explicitly*. You will not always call Xlib functions or structures explicitly (but recall that Motif and Xt are built upon Xlib and they may call Xlib function from their own function calls).

In order to distinguish between the various toolkits, X adopts the following convention:

- Motif function and data structure names begin with **Xm**. So in `push.c`:  
`XmStringCreateSimple()` and `XmStringFree()` belong to Motif toolkit.
- Xt Intrinsic functions and most data structures begin with **Xt**. *e.g.* `XtVappInitialize()` and `XtVaCreateManagedWidget()`. The `Widget` data structure is an exception to this rule.
- Xlib functions and most data structures begin with **X**. There are no Xlib functions used in `push.c`.  
 An example of an Xlib function call is `XDrawString()`.

## Header Files

In order to be able to use various Motif, Xt Intrinsic or Xlib data structures we must include header files that contain their definitions.

The X system is very large and there are many header files. Motif header files are found in `#include<Xm/...>` subdirectories, the Xt and Xlib header files in `#include<X11/...>` subdirectories (*E.g.* the Xt Intrinsic definitions are in `#include<X11/Intrinsics.h>`).

Every Motif widget has its own header file, so we have to include the `<Xm/PushButton.h>` file for the push button widget in `push.c`.

We do not have to explicitly include the Xt header file as `<Xm/Xm.h>` does this automatically. Every Motif program will include `<Xm/Xm.h>` -- the general header for the motif library.

## Compiling Motif Programs

To compile a Motif program we have to link in the Motif, Xt and Xlib libraries. To do this use: `-lXm -lXt -lX11` from the compiler command line. **NOTE:** The order of these is important.

So to compile our `push.c` program we should do:

```
cc push.c -o push -lXm -lXt -lX11
```

The exact compilation of your Motif programs may require other compiler directives that depend on the operating system and compiler you use. You should *always* check your local system documentation or check with your system manager as to the exact compilation directives. You should also check your C compiler documentation. For example you may need to specify the exact path to a nonstandard location of include (`-I flag`) or library (`-L flag`) files. Also our `push.c` program is written with ANSI style function calls and some compilers may require this knowledge explicitly. Some implementations of X/Motif do not strictly adhere to the ANSI C standard. In this case you may need to turn ANSI C function prototyping *etc.* off.

Having successfully compiled you Motif program the command:

```
push
```

should successfully run the program and display the PushButton on the screen.

## Basic Motif Programming Principles

Let us now analyse the `push.c` in detail. There are six basic steps that nearly all Motif programs have to follow. These are:

1. Initializing the toolkit
2. Widget creation
3. Managing widgets
4. Setting up events and callback functions
5. Displaying the widget hierarchy
6. Enter the main event handling loop

We will now look at each of these steps in detail.

### Initialising the toolkit

The initialisation of the Xt Intrinsics toolkit must be the first stage of any basic Motif program.

There are several ways to initialise the toolkit. `xtVaAppInitialize()` is one common method. For most of our programs this is the only one that need concern us.

When the `XtVaAppInitialize()` function is called, the following tasks are performed:

- The application is connected to the X display.
- The application is parsed for the standard X command-line arguments.
- Resources are set up.
- A top level window is created -- this is returned by the function call to the widget data structure `top_wid` in `push.c`.

`XtVaAppInitialize()` has several arguments:

### Application context

(address of) -- This is a structure that Xt requires for operation. For the Motif programs that we will be considering we do not need to know anything about this, except the need to set it in our program.

### Application class name

-- A string that is used to reference and set resources common to the application of even a collection of applications. Chapter 11 deals with many resource setting mechanisms that uses this class name. In these coming examples note that the class name has been set to the string, "Push".

### Command line arguments

-- The third and fourth arguments specify a list of objects of the special X command line arguments that can be specified to an X program. The third argument is the list, the fourth the number in the list. This is advanced X use and is not considered further in this text. Just set the third argument to `NULL` and the fourth to 0. The fifth and sixth arguments `&argc` and `argv` contain the values of any command line argument given. These arguments may be used to receive command line input of data in standard C fashion (e.g. filenames for the program to read). Note that the command line may be used (Section 11) to set certain resources in X. However these will have been removed from the `argv` list if they have been correctly parsed and acted upon before being passed on to the remainder of the program.

### Fallback Resources

-- Fallback resources provide security against errors in other setting mechanisms. Fallback resources are ignored, if resources are set by any other means. Chapter 11 deals with many resource setting mechanisms and Section 11 gives examples of setting fallback resources. A fallback resource is a `NULL` terminated list of strings. For now we will simply set it to `NULL` as no fallback resources have been specified.

### Additional Parameters

-- a `NULL` terminated list. These are also used only for advanced applications, so we will set them to `NULL`.

## Widget Creation

There are several ways to create a widget in Motif:

- There is a specific function for creating each widget.
- There are several convenience functions for *generic* widget creation and even creating and managing widgets with a single function call.

We will introduce the convenience functions shortly but for now we will continue with the simpler first method of widget creation.

In general we create a widget using the function:

```
XmCreate<widget name>().
```

So, to create a push button widget we use `XmCreatePushButton()`

Most `XmCreate<widget name>()` functions take 4 arguments:

- The parent widget -- `top_wid` in `push.c`.
- The name of the created widget -- a string - `"Push_Me"` - in `push.c`.
- Command line / Resource list -- `NULL` in `push.c`.
- The number of arguments in the list.

The argument list can be used to set widget resources (height, width *etc.*) at creation. The name of the widget may also be important when setting a widget's resources. The actual resources set depend on the class of the widget created. The individual Chapters and reference pages on specific widgets list widget resources. Chapter [10](#) deals with general issues of setting resources and explores the methods described here further.

## Managing Widgets

Once a widget has been created it will usually want to be managed. `XtManageChild()` is a function that performs this task.

When this happens all aspects of the widget are placed under the control of its parent. The most important aspect of this is that if a widget is left unmanaged, then it will remain *invisible* even when the parent is displayed. This provides a mechanism with which we can control the on screen visibility of a widget -- we will look at this in more detail in Chapter [10](#). Note that if a parent widget is not managed, then a child widget will remain invisible *even if* the child is managed.

Let us leave this topic by noting that we can actually create and manage a widget in one function called `XtVaCreateManagedWidget()`. This function can be used to create any widget. We will meet this function later in the Chapter [10](#).

## Events and Callback Functions

### Principles of Event Handling

When a widget is created it will automatically respond to certain internal events such as a window manager request to change size or colour and how to change appearance when pressed. This is because Xt and Motif

frees the application program from the burden of having to intercept and process most of these events. However, in order to be useful to the application programmer, a widget must be able to be easily attached to application functions.

Widgets have special *callback functions* to take care of this.

An *event* is defined to be any mouse or keyboard (or any input device) action. The effect of an event is numerous including window resizes, window repositioning and the invoking functions available from the GUI.

X handles events *asynchronously*, that is, events can occur in any order. X basically takes a continuous stream of events and then dispatches them according to the appropriate applications which then take appropriate actions (remember X can run more than one program at a time).

If you write programs in Xlib then there are many *low level* functions for handling events. Xt, however, simplifies the event handling task since widgets are capable of handling many events for us (e.g. widgets are automatically redrawn and automatically respond to mouse presses). How widgets respond to certain actions is predefined as part of the widget's resources. Chapter 17 gives a practical example of changing a widget's default response to events.

## Translation tables

Every widget has a *translation table* that defines how a widget will respond to particular events. These events can enable one or more actions. Full details of each widget's response can be found in the Motif Reference material and manuals[Hel94b].

An example of part of the translation table for the push button is:

```
bBtn1downn:    Activate(), Disarm() BSelect Press:    Arm()
BSelect Click:    Activate(), Disarm()
```

BSelect Press corresponds to a left mouse pressed down and the action is the Arm() function being called which causes the display of the button to appear as it was depressed. If the mouse is clicked (pressed and released), then the Activate() and Disarm() functions are called, which will cause the button to be reactivated.

Keyboard events can be listed in the table as well to provide facilities such as *hot keys*, function/numeric select keys and help facilities. These can provide short cuts to point and click selections.

Examples include: KActivate -- typically the return key, KHelp -- the HELP or F1 key.

## Adding callbacks

The function Arm(), Disarm() and Activate() are examples of predefined *callback functions*.

For any application program, Motif will only provide the GUI. The Main body of the application will be attached to the GUI and functions called from various events within the GUI.

To do this in Motif we have to add our own callback functions.

In `push.c` we have a function `pushed_fn()` which prints to standard output.

The function `XtAddCallback()` is the most commonly used function to attach a function to a widget.

It has four arguments:

- The widget in which the callback is to be installed, `button` in our example.
- The name of the callback resource. In our example we set `XmNactivateCallback`.
- The pointer to the function to be called.
- Client data that may get passed to the callback function. Here we do not pass any data and it is therefore set to `NULL`.

In addition to performing a job like highlighting the widget, each event action can also call a program function. So, we can also hang functions off the `arm`, `disarm` *etc.* actions as well. We use `XmNarmCallback`, `XmNdisarmCallback` names to do this.

So, if we wanted to attach a function `quit()` to a `disarm` for the `button` widget, we would write:

```
XtAddCallback(button, XmNdisarmCallback, quit, NULL);
```

## Declaring callback functions

Let us now look at the declaration of the application defined callback function. All callback functions have this form.

```
void pushed_fn(Widget w,
               XtPointer client_data,
               XmPushButtonCallbackStruct *cbs)
```

The first parameter of the function is the widget associated with the function (`button` in our case).

The second parameter is used to pass client data to the function. We will see how to attach client data to a callback later. We do not use it in this example so just leave it defined as above for now.

The third parameter is a pointer to a structure that contains data specific to the particular widget that called the function and also information about the event that triggered the call.

The structure we have used is a `XmPushButtonCallbackStruct`. A *Callback Structure* has the following general form:

```
typedef struct {
    int reason;
    XEvent *event;
    .... widget specifics ... } Xm<widget>CallbackStruct;
```

The `reason` element contains information about the callback such as whether `arm` or `disarm` invoked the call and the `event` element is a pointer to an (Xlib) `XEvent` structure that contains information about the

event.

## Finishing off -- displaying widgets and event loops

We have nearly finished our first program. We have two final stages to perform which every Motif program has to perform. That is to tell X to:

- Display or **realize** the widgets . This is achieved via the `XtRealizeWidget()` function . In `push.c` we pass the top level widget `top_wid` to the function so that all child widgets are displayed.
- Enter the main event handling loop . The function `XtAppMainLoop(app)` does this. After this call, Xt has control over the program and it is this that dispatches events that invoke callbacks *etc.* **Note:** The application code will be idle until a user activates an event.

## Exercises

### Exercise 8295

Write a Motif program that displays a button labelled "Quit" which terminates the program when the button is depressed with the left mouse button.

## Widget Basics

In the last Chapter we introduced some basic Motif programming concepts and introduced one specific widget, the `PushButton`, used in the example program developed. There are many other classes of widgets defined in Motif. Motif widgets are provided to perform a wide variety of common GUI tasks. This Chapter overviews the classes of Motif widgets. Following Chapters go on to study individual widgets in detail.

## Widget Classes

The organisation of a large system of GUI components in Motif can be quite complex. In order to aid the design and understanding of Motif various widget classes have been constructed. Each class can be categorised by a broad functionality, at a variety of levels. Motif defines a hierarchy of widget classes (Fig 6.1) and a widget will *inherit* properties from a higher class in the widget hierarchy (Section 6.5). Some levels of the hierarchy have strong relationships with Xt Intrinsics since widgets are actually created in this toolkit.



**Fig. 6.1 Widget Hierarchy** The levels of the hierarchy and a broad functionality of each level are:

### Core

-- The top of the hierarchy comes from Xt intrinsics. This *superclass* defines background, size and position properties that are common to all widgets.

### XmPrimitive



-- The superclass of all primitive widgets. Primitive widgets are the basic building blocks of any Motif GUI (See Section [6.1.3](#) below).

### Composite

-- The superclass of containers for widgets. Two sub-classes of the *Composite* class are defined:

#### Shell

widgets control the interfacing of Motif with the window manager.

#### Constraint

widgets are concerned with the organisation (positioning, alignment, *etc.*) of widgets contained within them. *XmManager* is a subclass of the constraint widget class. Manager widgets are discussed further in Section [6.1.3](#) below.

## Shell Widgets

All widgets are contained in a shell widget. This is usually the top level widget. The primary function of a shell widget is as an interface to the window manager.

The **application** shell is normally the top level for an application and is created by `XtVaAppInitialize()` or related functions.

There are two other shells:

- The **Override shell** is used for pop-up menus that must be at the top-level. To achieve this, the window manager must usually be bypassed. Consequently, the override shell is not often used by Motif programs.
- The **Transient** shell is used for dialogs. However Motif repackages this shell so that dialogs become a subclass widget.

## Constraint Widgets

Constraint widgets are concerned with the positioning and alignment of widgets contained within them. *XmManager* is a (Motif) subclass of the constraint widget specifying general manager facilities concerned with, for example, callbacks and highlight colour.

## Construction widgets

Motif defines two basic classes of widgets that provide the basic building blocks of any GUI: **primitive** and **manager**. The majority of the remainder of this text is devoted to these widget classes.

Within each of these basic classes, several different sub-classes of widget are defined.

The broad function of each class is as follows:

### Primitive

widgets are designed to work as a single entity. They provide the building blocks with which we assemble our GUI. The `PushButton` is an example of a primitive widget class.

### Manager

widgets are designed to be *containers* and may have primitive and manager widgets placed under their control. The main function of manager widgets is to help control the design of a GUI. Manager widgets control how we organise a GUI by prescribing standard, or uniform, layouts (such as the `MainWindow` widget, Chapter 9) or providing widgets that let us place widgets in an ordered fashion (e.g. `RowColumn` or `Form` widgets, Chapter 8).

Following Chapters will give details and examples of all types of widgets. For the remainder of this chapter we will give a brief introduction to these widgets.

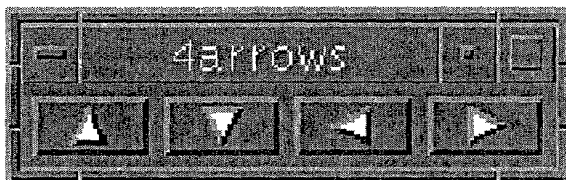
## Primitive Widgets

The following primitive widgets are defined in Motif:

### ArrowButton

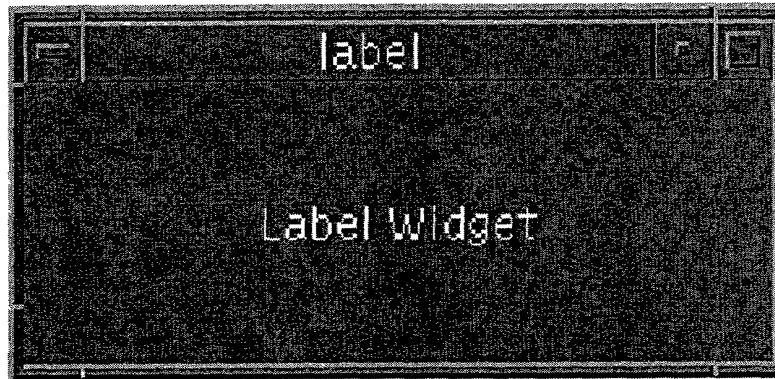
-- A button with an orientatable arrow (Fig. 6.2). This button is defined and used in a similar fashion to the `PushButton` widget. An example of the `ArrowButton` can be seen in the `arrows.c` program in Chapter 8.

**Figure 6.2:** The Four Orientations of the ArrowButton Widget



### Label

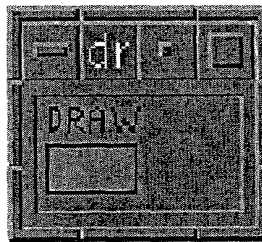
-- A widget which has text or an image (Pixmap) associated with it (Fig 6.3). The basic `Label` widget, as its name implies, does not do anything interactively. Its sole purpose is to help facilitate visual aids within the GUI. The `Label` widget does, however, have four (interactive) sub-classes of button:

**Figure 6.3: A Label Widget****PushButton**

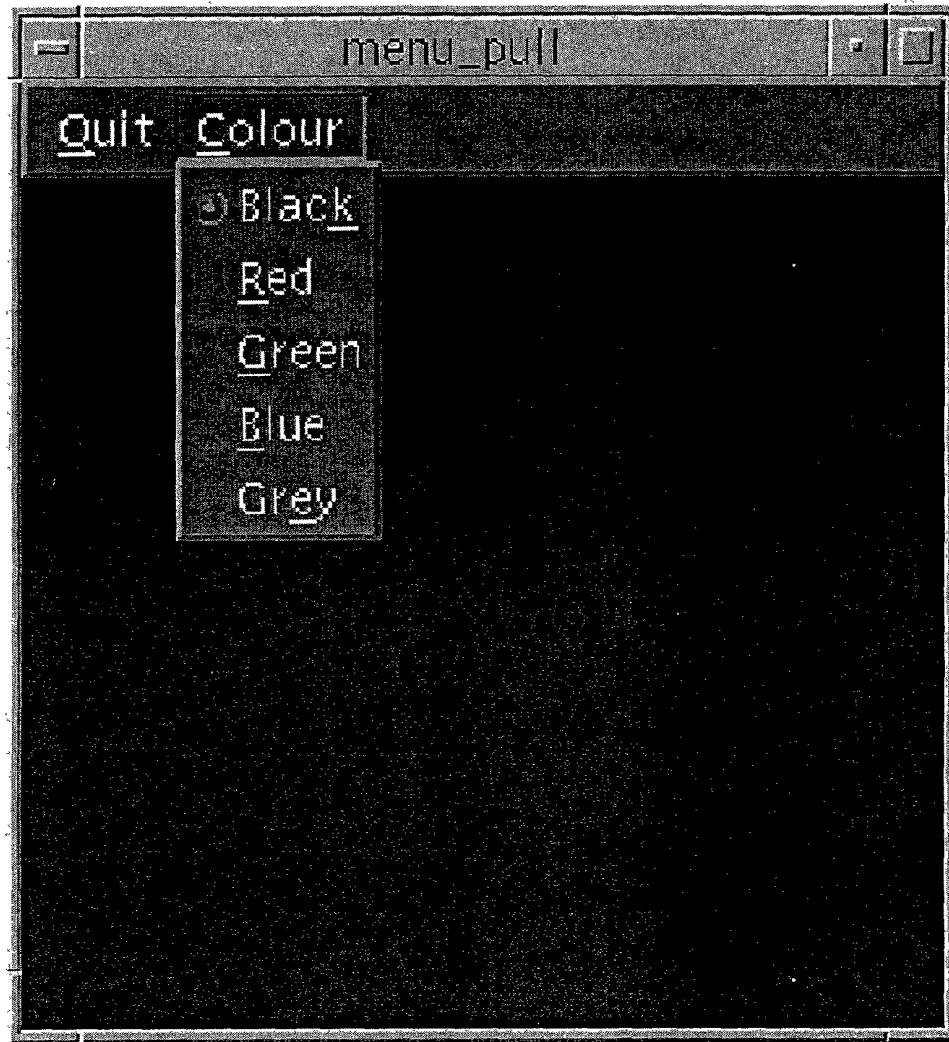
-- A button that can be labelled with a String (Fig 5.1). We have already met this widget in our first program, `push.c` (Chapter 5).

**DrawnButton**

-- A button with which an icon (Pixmap) can be associated (Fig. 6.4).

**Figure 6.4: A  
DrawnButton Widget****CascadeButton**

-- A button usually associated with a PullDown menu (Fig. 9.2). This widget is described in association with PullDown menu widgets in Chapter 9.

**Figure 6.5:** A CascadeButton Widget and Associated PullDown Menu**ToggleButton**

-- A button that displays text or graphics together with a graphic indicator of the state of the ToggleButton. The ToggleButton has two states: either *on* or *off*. Toggle buttons may be grouped together to provide a variety of configurations. *RadioBox* are groups of ToggleButtons, where only one button can be selected at a time. A *CheckBox*, on the other hand, allows any number of buttons to be selected at a given time (Fig. 6.6). Examples of both configurations of ToggleButton are given in Chapter 15.

**Figure 6.6: A RadioBox and CheckBox ToggleButton Widget Configuration**



### Scrollbar

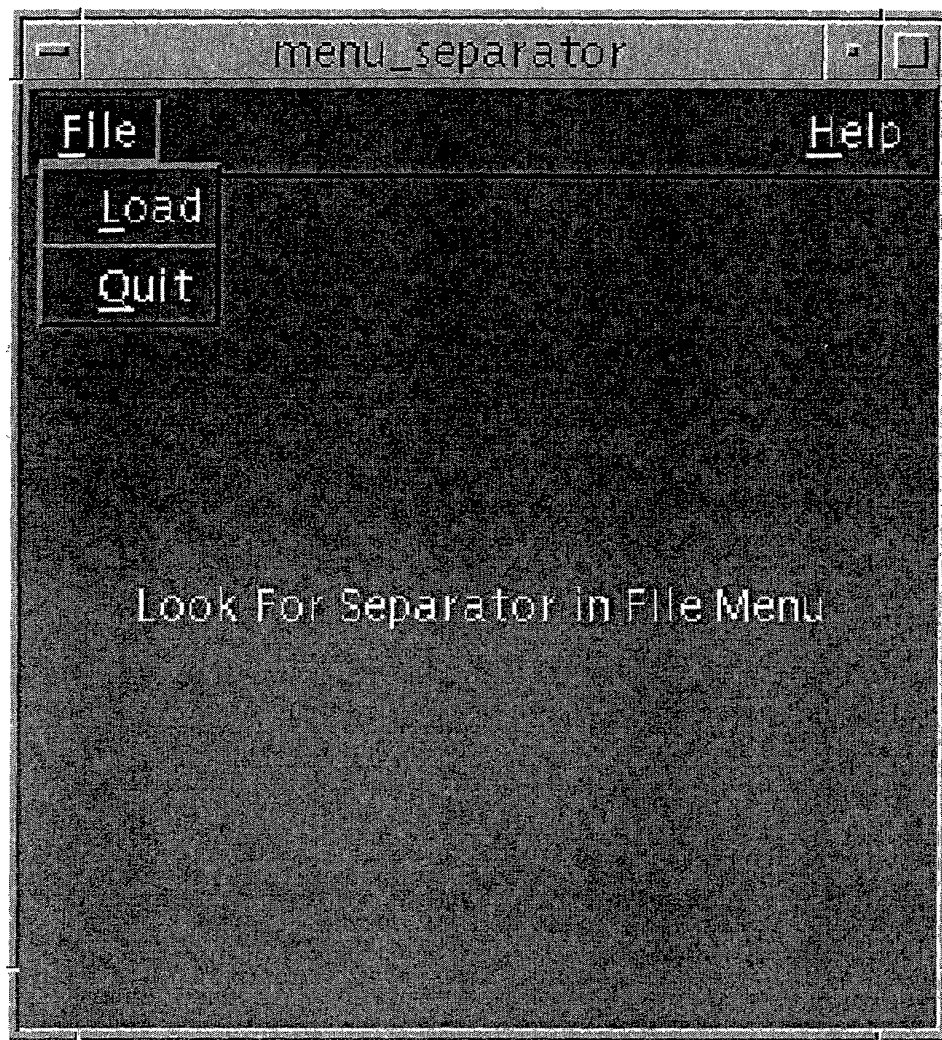
-- A widget that allows the contents of a window to be displayed in a reduced area where the user can *scroll* the window to view hidden parts of the window. This widget can be defined and used explicitly in Motif programs (Chapter 14). Frequently, scrolling control will be defined when the ScrollBar (Fig. 6.7) is *compounded* with another widget. Chapters 11 and 17 discuss examples of the Scrollbar used in conjunction with Text (Fig. 6.10) and DrawingArea widgets respectively.

**Figure 6.7: A Scrollbar Widget**



### Separator

-- A widget used to separate items in a GUI to aid visual display. Fig. 6.8 illustrates the use of the *Separator* widget to delineate items (*Load* and *Quit*) in a single pulldown menu.

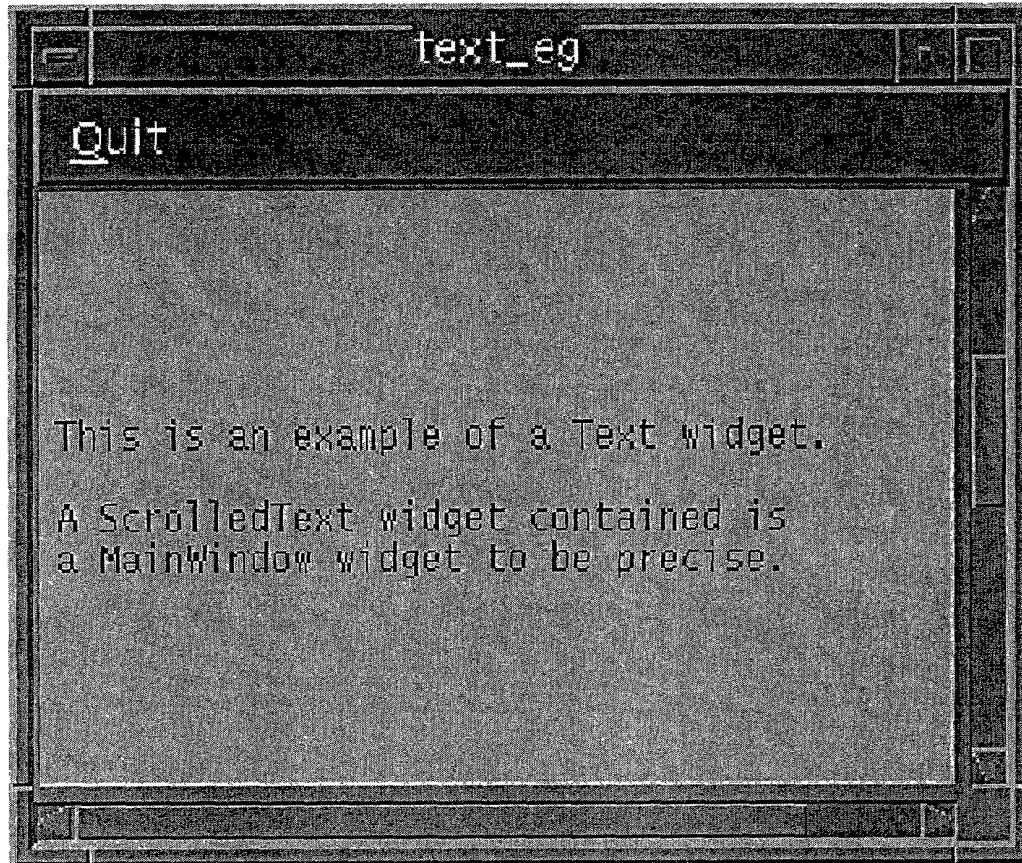
**Figure 6.8:** A Separator Widget Between Two Menu Items**List**

-- A widget that allows selection from a list of text items (Fig. 6.9). This widget is described in Chapter 12.

**Figure 6.9:** A List Widget**Text**

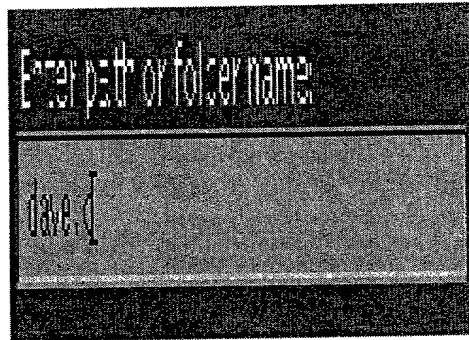
-- A complete text editor widget (Fig. 6.10). This widget provides default callback resources for text selection, editing, cutting and pasting of text as well the editing style of the widget. Additional callback resources can easily be attached to extend and customise the text widget for many text editing applications. Chapter 11 addresses all such issues.

Figure 6.10: A Text Widget



### TextField

-- A single-line text editor. This is basically a Text widget which is limited to a single-line of text entry. The TextField widget has many uses in GUI design where simple input is required. For example, when a file name needs to be specified as input in a GUI or a key word is required for some search.

**Figure 6.11: A TextField Widget**

Motif 2.0 also prescribes another form of text widget, **CSText** . This widget provides the same facilities as the Text widget but uses an alternative (compound) text string representation, `xmString` (Section 6.6), which is capable of supporting multiple fonts .

## Gadgets

There may be occasions in Motif programming when we want to have the properties of a primitive widget but do not wish to worry about the management of the window properties of the widget. Motif has to manage each created widget's window and associated resources. If we have several widgets within our interface this could cause complications for our application.

To attempt to alleviate many of these problems Motif provides *Gadgets* . Gadgets are basically windowless widgets and, therefore, require less resources than a widget. Control of the gadget is the responsibility of the parent of the gadget.

Not all widgets have corresponding gadgets. The following gadgets are available: **ArrowButton**, **Label** and **Separator**. They behave in a similar manner to their corresponding widgets.

We will not consider gadgets further in this book since our programs are relatively simple and the relevance of their use cannot be effectively illustrated. The primary use of gadgets is when there is a real need to save memory on the X server or within the application. Gadgets may actually increase computer processor load since X finds some events more difficult to track within them. Gadgets are basically an artefact from earlier versions of Motif that were developed when window construction and management was more critical. Later versions of X have optimised the X server and coupled with the fact that computer power has increased and memory is less expensive, the use of gadgets is not as important as it once used to be.

## Manager Widgets

Manager widgets are the basic Motif widgets for constructing and organising our interfaces in Motif. The following *manager widget* classes are available:

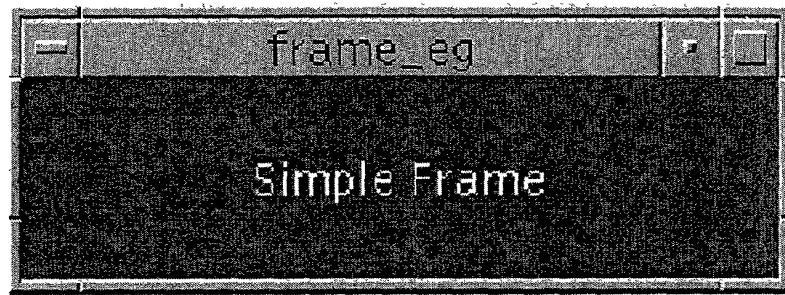
### Frame

-- A widget that provides an embossed effect to the *child* widget it contains (Fig 6.4). This is the



simplest manager widget. An example of this widget is given in `frame.c` (see Exercise 6.1).

**Figure 6.12: A Frame Widget**



### ScrolledWindow

-- A widget that allows scrolling of its child widget. Fig. 6.10 illustrates a common example of this: a *ScrolledText* widget that has a Text widget capable of being scrolled in a horizontal and vertical direction.

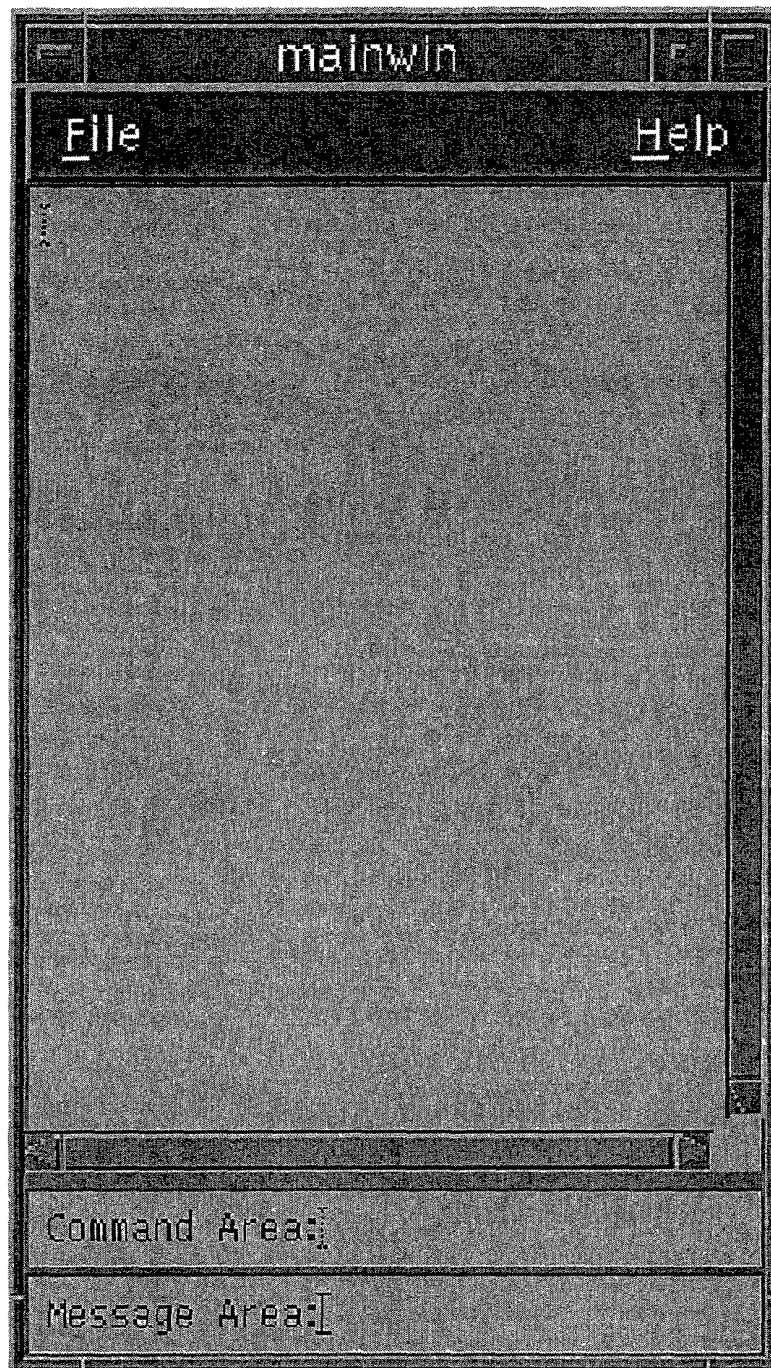
One common subclass of the ScrolledWindow widget is the MainWindow widget:

### MainWindow

-- A typical *top level* container widget for an application. This widget provides a common, uniform *look and feel* for any GUI (similar to MS Windows *look and feel* (Section 3.1)). The MainWindow widget has well defined mechanisms for the provision of Menubars, Scrollbars and command and message windows (Fig. 6.13). Chapter 9 describes the major aspects of MainWindow programming.

**Figure 6.13: A MainWindow Widget**

000260" 20284960



### DrawingArea

-- A widget where graphics can be displayed (Fig. 6.14). **Note:** Motif does not provide any graphics functions, Xlib provides all the graphic drawing and manipulation routines. Graphic programming and the interface with Xlib is one of the more difficult aspects of Motif to understand. Chapters 16--18 discuss these more advanced issues of programming and show how the DrawingArea widget is used in practice.

**Figure 6.14:** A DrawingArea Widget



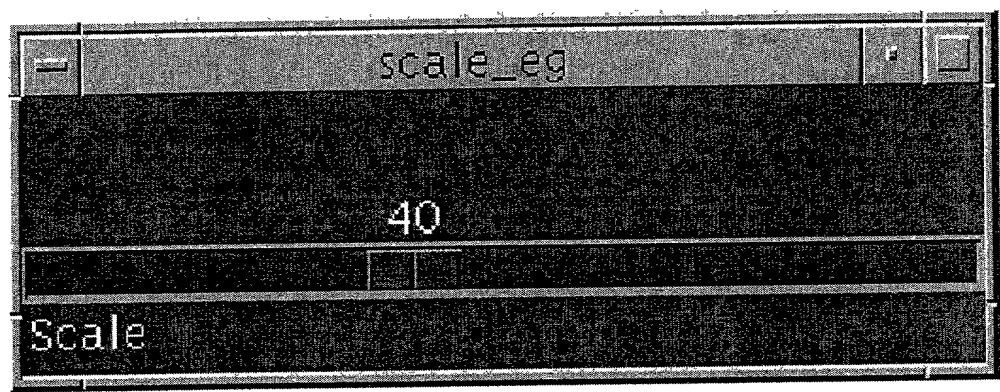
### **PanedWindow**

-- Allows vertical tiling of child widgets. The use of this widget is not as common as other widgets and we will not address its use on this particular journey through Motif.

### **Scale**

-- This widget provides a *slider* object that can be used for user input (Fig. 6.4). Chapter 13 describes the Scale widget.

Figure 6.15: A Scale Widget



- RowColumn
- A widely used widget that can lay out widgets in an orderly 2D fashion. Chapter 8 describes this widget.
- BulletinBoard
- There are two sub-classes of this widget:
- Form
- A widget similar in use to *RowColumn* but allows greater control of the placement and sizing of widgets. Chapter 8 compares and contrasts the *Form* and *RowColumn* widgets.
- Dialog
- There are two forms of dialog:
  - a *MessageBox* which simply gives information to the user (Fig. 6.16) and
  - a *SelectionBox* which allows interaction with the user. Motif provides two sub-classes of the *SelectionBox* widget: a *Command* widget for command line type input and a *FileSelectionBox* for directory/file selection (Fig. 6.17).

Figure 6.16: A MessageBox (ErrorDialog) Widget

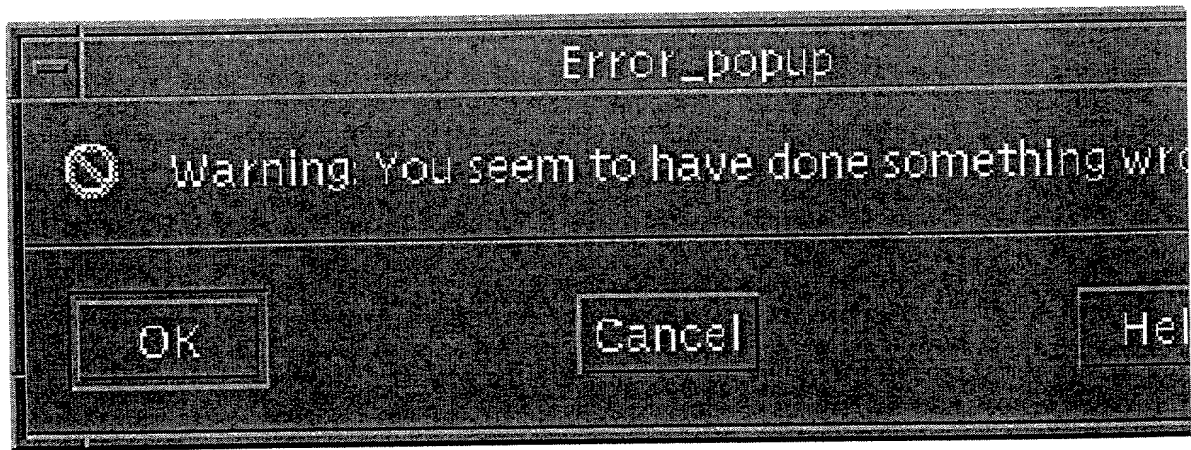
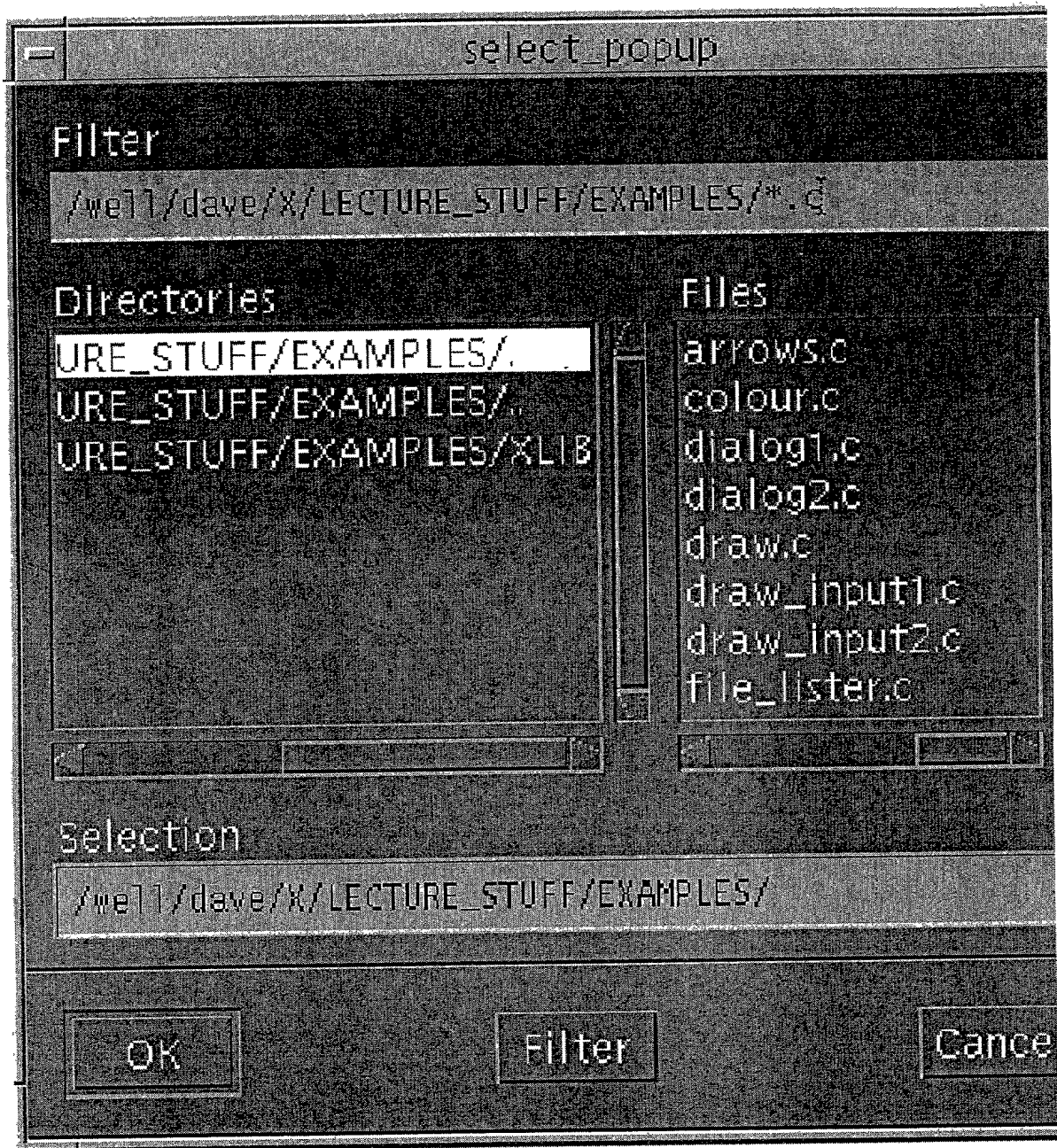


Figure 6.17: A FileSelectionBox Widget





Dialog widgets, as the name implies, provide the direct line of communication between the user and the application. In the case of the *MessageBox* widgets this could simply be the program informing the user of some event or action. There are several prescribed classes of *MessageBox* widgets that are associated with a special event. For example there are *WarningDialog*, *InformationDialog* and *ErrorDialog* widgets (Fig. 6.16). Also some form of prescribed user interaction is provided by the *Command* and *FileSelectionBox* widgets. Motif also provides base *MessageBox* and *SelectionBox* widgets so that the programmer can assemble customised Dialogs. However, Motif programming style (Chapter 20) suggests that wherever appropriate the prescribed Dialog widgets should be used to provide uniformity across applications. Chapter 10 deals with many aspects of Dialog widget programming and usage.

## Motif 2.0 Widgets

Motif 2.0 defines a few new Manager widgets:

### **ComboBox**

-- A widget that combines the capabilities of a single line TextField and an XList.

### **IconGadget**

-- A widget that can be used to display Icons.

### **Container**

-- A widget that manages IconGadget children.

### **Notebook**

-- A widget that organizes children into pages, tabs, status area and page scroller.

### **Scale (thermometer)**

-- A modified version of the Scale widget with new resources added for thermometer behavior (see Chapter 13).

### **SpinBox**

-- A widget that manages multiple traversable children.

The use of many of these new widgets is fairly advanced and, except where indicated, these widgets are not dealt with further in this introductory text. Chapter 21 gives some further details on Motif 2.0.

## **Widget Resources**

Each widget has a number of resources. These control many features of the widget such as the foreground and background colours, size *etc.*. A particular widget will have specialised resources such as callback resources which define how the widget responds to an event *etc.*.

Every widget is documented in the Motif Reference Manual ([Hel94b]) which gives a complete list of the resources that a particular widget employs. When discussing individual widgets we will only consider the important resources that define the main characteristics of the widget concerned. The following Chapter addresses how widget resources can be set and altered for a given application.

There is a hierarchy of widgets (Fig 6.1) and a widget will *inherit* resources from a higher resource class in the widget hierarchy.

The levels of the hierarchy and related widget resources are:

### **Core**

-- This *superclass* gives background, size and position resources that are common to all widgets.

### **XmPrimitive**

-- The superclass of all primitive widgets defines resources related areas such as foreground and highlight colour.

### **Composite**

-- The superclass of containers for widgets has two sub-classes:

#### **Shell**

widgets have resources related to interfacing with the window manager.

#### **Constraint**

widgets have resources that are concerned with the positioning and alignment of widgets contained within. *XmManager* is a subclass of constraint and specifies general manager widget resources such as callbacks and highlight colour.

### **Widget level**

-- Resources particular to a specific widget.

# Strings in Motif

Motif programs (in C/C++) will typically need to use both types of *string* available:

- A data type `String` -- A "normal" C string (an array of characters). However, for convenience Motif has defined `String` as a *standard* Motif Data Type. Clearly, as Motif programs are usually written in C, this data type will be the main means of communication between a program and the standard input and output mechanisms.
- A data type `XmString` -- Motif *internal* string data structure. When Motif needs to *draw* strings on the display, to achieve this more information is needed than simply the array of characters. Consequently, Motif defines an `XmString` with a more complicated structure. Most widgets can have a label resource associated with them, which is usually an `XmString` data type. It is rarely that the Motif programmer will need to know the *internal* structure of an `XmString`.

Motif provides a number of functions to convert a `String` into an `XmString` or vice versa:

`XmStringCreateLocalized()`, `XmStringCreateLtoR()`, `XmStringGetLtoR()`, `XmStringCompare()`, `XmStringConcat()`, `XmStringCopy()`, *etc.* are common examples. Many behave in a similar manner to their C standard library string handling function counterparts.

Their use is fairly straightforward -- the reference manuals should be consulted for more details.

## Exercises

### NEED SOME MORE

#### Exercise 8395 (1)

Run the following program (`frame.c`) and note the difference in appearance and interaction of the widget with the `push.c` program (Chapter 5). Note the effect of the `XmNshadowType` resource. What other settings are available for this resource and what effect do they have?

```
/* frame.c --
   mount pushbutton of push.c in a frame widget
   */

#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Frame.h> /* header file for frame stuff */

/* Prototype callback */

void pushed_fn(Widget , XtPointer ,
               XmPushButtonCallbackStruct *);
```

```

main(int argc, char **argv)
{
    Widget          top_wid, button, frame;
    XtAppContext app;

    top_wid = XtVaAppInitialize(&app, "Push", NULL, 0,
                               &argc, argv, NULL, NULL);

    frame = XtVaCreateManagedWidget("frame",
                                     xmFrameWidgetClass, top_wid,
                                     XmNshadowType, XmSHADOW_IN,
                                     NULL);

    button = XmCreatePushButton(frame, "Push_me",
                                NULL, 0);

    XtManageChild(button);

    XtAddCallback(button, XmNactivateCallback, pushed_fn, NULL);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

void
pushed_fn(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)
{
    printf("Don't Push Me!!\n");
}

```

### Exercise 8397

Rewrite the `frame.c` program (Exercise 6.1) so that it displays a button labelled ``Quit Frame" which terminates the program when the button is depressed with the left mouse button.

### Exercise 8398

Write a program that inputs the string ``X Convert Me!!" as a standard `String` data type and converts the string to an `XmString` data type. The program should also extract the substring "Convert Me!!" from the `XmString` and store it as a `String`.

## Widget Resources

Every Motif widget has a number of resources that control or modify its behaviour and appearance. Depending on the widget's class, resources control aspects such as the size of the widget, the colour of the widget, whether scroll bars should be displayed and many other properties.

When a widget is created it inherits resources from a higher widget class and also creates its own (Section 6.5). All these resources have default values. It would be tedious to specify 30 plus resource values each time we create a widget.



However the application programmer or user may want to customise one or two resources in order to control a widget's size in a GUI or its dimensions on the screen, for example.

Throughout this Chapter we will use the PushButton program, `push.c`, developed in Chapter 5 as a case study and we shall see how we alter the size (width and height) of the button. The resource variables `XmNwidth` and `XmNheight` hold these values.

There are a few ways in which we can alter a particular resource's value.

## Overriding Resource Defaults

Broadly, there are two methods of altering resources in Motif:

- External resource files -- resource values are set and stored in particular files.
- Hard Coding in the program -- resource values are set in the program code.

The advantage of using external files is that it allows the user to customise Motif applications without having to recompile the program. The user may never need to access the source code. Applications may also be customised each time they are run. Due to the wide variety of possible systems running X and a vast range of user needs this can be a useful feature. For example, screen resolutions vary a great deal from device to device and what may be a visually adequate size for a widget on one display may be totally inadequate on another. Also certain resources, particularly colour displays, may vary substantially across different platforms.

There are four basic ways to externally customise an application:

### User resource file

- a file placed in the users home directory called `.xdefaults`.

### Class resource file

- a single application can have a special file reserved for its particular resource setting commands.

### Command line parameters

- a widget's resources can be specified when the program is actually run.

### RESOURCE\_MANAGER/ SCREEN\_RESOURCES properties

- A standard X Window program, `xrdb`, can be used to set certain application resources.

One problem with the external setting of resource values occurs if it has already been hard-coded, since hard-coded resource values have a higher precedence than all externally set resources.

There are some advantages to hard-coding resource values :

- It may be essential to prevent a user changing particular resource values. For example, changing certain widget sizes may totally disorganise the GUI design and display.
- Resource files require specific names and locations that may be hard to install and maintain correctly. File and directory permissions may occasionally be set so as to make the files unreadable. Files and directories sometimes get deleted or moved.
- Unix environment variables sometimes cause conflicts with external resource value settings.

A trade-off is sometimes required between applying hard-coding and allowing user freedom.

There is one other internal coding method for changing resources, by using what are called *fallback resources*. As their name implies fallback resources are set only if a resource has not been set by any of the above means. Fallbacks are therefore useful for setting alternatives to the default Motif resource settings.

The remaining sections in this Chapter detail how each individual resource setting method may be used.

## User Resource (.xdefaults) File

When Motif initialises the application it looks for the .xdefaults file in your HOME directory. The .xdefaults file is a standard text file, where each line may contain a resource value setting, a blank line or a comment denoted by an exclamation mark (!). This file contains directives that can reset any resource for any application, using the following method:

- The resources relating to widgets created within a particular application will be referred to by the application class name set when the `XtVaAppInitialize()` is called (Section 5.6.1). The application class name was set to "Push" in our example. Application names can also be used to refer to resources, but this is not as common as the class name. The application name is the name of the compiled program, push in this example.
- The resource name for a particular widget will have been specified in the program via the name argument of the widget creation function (Section 5.6.2). The name of the PushButton in our example is "Push\_me". **Note:** this is not the variable name but the name argument in the create widget function.
- The resource value is then accessed via:

```
application_class_name.widget_name.resource_name.
```

Note that when referring to resource names outside a program, the `xmN` part of the resource is dropped from the resource name. So, the width and height of the PushButton widget in the `push.c` application are referred to by:

```
Push.Push_me.width OR Push.Push_me.height
```

- The directive in the file to set the width and height of the PushButton widget of `push.c` is of the form:

```
! Comment: Set push application, PushButton dimensions
```

```
Push.Push_me.width: 200
Push.Push_me.height: 300
```

where 200 and 300 are the widget's new dimensions. Note that there is a colon separating the resource name and its new value.

Wild card (\*) settings are also allowed. Therefore to set *all* widgets called ``Push\_me" width and height you could write:

```
! Comment: wildcard setting of widget Push_me dimensions

*Push_me.width: 200
*Push_me.height: 300
```

Motif widget class names may also be used. This will set *all* widgets of a given class and application to the same values so care should be taken. An alternative method to set the width and height of the PushButton of push.c is to use the Motif XmPushButton class name to set its resources via:

```
! Comment: Set push application,
! XmPushButton widget class dimensions

Push*XmPushButton.width: 200
Push*XmPushButton.height: 300
```

## Class Resource File

The class resource file relates to a particular application, or class of applications. The application class name created with XtVaAppInitialize() is used to associate a class resource file with an application. Thus, the push.c program has an application class ``Push" and therefore the application would have a class file named Push. The class resource files are normally stored in the user's home directory.

It is possible, and quite practical, to associate several applications with a single class name (by setting the appropriate XtVaAppInitialize() argument) and therefore with a single class resource file. This would allow for one class file to control the resource setting for many usually similar applications. Individual applications can again be referred to by their (compiled) program name, but this is not common.

The setting of individual resource values is as described for the .Xdefaults file, except that the wild card matching may not have such far reaching consequences. Therefore, to set the dimension of the PushButton in push.c we could write:

```
! Class Resource File for push.c called "Push"
! Store in HOME directory

*Push_me.width: 200
*Push_me.height: 300
```

## Command Line Parameters

Resource values can be set with X window command line parameters. Some common resources can be easily referred to and a general command exists to set any resource value. The advantage of using the command line is that resource values can be altered each time the program is run. This is ideal if you only change a resource infrequently, or you are experimenting to find suitable resource values. However, setting resource values in this fashion regularly involves a lot of typing, so alternative methods should be considered.


Common resource values have special abbreviations for command line operation. These are listed in Table. 

Table: Command Line Resource Options

Option	Data Passed	Use
-bg	background colour	Sets background colour
-fg	foreground colour	Sets foreground colour
-geometry	WidthxHeight+Xorigin+Yorigin	Sets window dimensions and location
iconic		Starts application as an icon
-name	Application name	Sets instance name not class
-title	Application title	Sets window title bar name
-display	Display name	Sets X server connection
-xnl language	language	Sets internationalisation locale
-xrm	X resource manager string	Set any other resources

The foreground and background colours are referred to by the common colour name database which is detailed in the reference manuals ([Hel94b], Section 18.4). So to set the background to red for the push.c program. We would run the program from the command line as follows:

```
push -bg red
```

The -geometry option sets the windows dimension and position. It has 4 parameters:

```
WidthxHeight+Xorigin+Yorigin
```

where width, Height, Xorigin and Yorigin are integer values separated by x or +.

Therefore to set the window size to 300 by 300 with the origin at top left corner run the program from the command line with the following arguments:

```
push -geometry 300x300+0+0
```

You can omit either the size, widthxHeight, or position, +Xorigin+Yorigin parameters in order to either size or position a window.

Therefore to set the window size to 500 by 500 and to allow the window manager to place it somewhere type:

```
push -geometry 500x500
```

and to explicitly position a window and use the default dimension type:

```
push -geometry +100+100
```

The -xrm option allows the user to set resources that are not otherwise facilitated from the command line. Following the -xrm option you supply a string that contains a resource setting similar to a single line resource value setting in a user or class resource file (including wildcards).

Therefore to change the highlight colour of the PushButton in push.c we could type:

```
push -xrm "Push*highlightColor: red"
```

where Push is the application class name and (XmN)highlightColor is the resource being set to red.

Multiple settings of resource values require -xrm calls for each resource value setting. Therefore to set the highlight and foreground colours for the above PushButton we could type:

```
push -xrm "Push*highlightColor: red" \
      -xrm "Push*foreground: blue"
```

# The Resource Manager Database

The X system provides a program, *xrdb*, that allow you to set up and edit resources stored on the root window of a display. The data is stored in `RESOURCE_MANAGER` property and, also, in `SCREEN_RESOURCES` property if multiple screens are supported. For the sake of simplicity we will assume that we are only working with a single screen and therefore do not need to consider the `SCREEN_RESOURCES` property further.

When *xrdb* is run it reads a `.Xresources` file from which it creates the `RESOURCE_MANAGER` property. The `.Xresources` file is usually stored in the users home directory. The `.Xresources` file can contain similar resource value settings as described for the `.Xdefaults` file and class resource file. However, *xrdb* can actually run the `.Xresources` or an other input file through the C preprocessor, so C preprocessor syntax is allowed in the `.Xresources` file. Several macros are defined so that constructs like `#ifdef` and `#include` may be used. One common example of their use is to set separate colour and monochrome resources for different screen settings. The macro `COLOR` is defined if a colour screen is present for a given display. Therefore, an `.Xresources` file could look for this and take appropriate actions:

```
#ifdef COLOR
! Colour screen detected set colour resources

Push*foreground: RoyalBlue
Push*background: LightSalmon

#else
! must be a monochrome screen

Push*foreground: black
Push*background: white
#endif
```

For further information on *xrdb* readers should consult the X Window system user reference manuals[QO90], or online manual documentation.

One advantage of this method of setting resources is that it allows for dynamic changing of defaults without editing files. In fact, the setting of resources via files may not work on X terminals with limited computing power, or when programs are run on multiple machines (since other machines may not have the appropriate `.Xdefaults` or class resource files). Having all the resource information in the server means that the information is available to all clients.

## Hard-coding Resources Within a Program

There are two basic methods for setting resource values from within a C program. The first method described below is dynamic, meaning that resources can be set and altered at any occasion from within the program. Another method allows resources to be set when a widget is created. Both these methods would

override other methods of resource setting.

## Dynamic control of resources

Using this method we can change the values of a resource at any time from within a program. Typically two functions are employed to do this:

- We use `XtSetArg()` to place resource values in an `Argument` list.
- `XtSetValues()` then sets these values in the list for a given widget. We need to specify the widget whose resources we wish to set, the `arg` list and the number of arguments in the list as parameters to this function.

Therefore, to set the size of the `button` resources in `push.c` we would use `XtSetArg()` to set width and height values in the `arg` list and then set the `arg` values for the `button` widget as follows:

```
Arg args[2]; /* Arg array */
int n = 0; /* number arguments */

XtSetArg(args[n], XmNwidth, 500);
n++;
XtSetArg(args[n], XmNwidth, 750);
n++;
XtSetValues(button, args, n);
```

A related function `XtGetValues()` exists to find out resource values (Chapters [12](#) and [14](#) contain some examples of `XtGetValues()` in use).

A convenience function `XtVaSetValues()` actually combines the above two operations and make programming a little less tedious. `XtVaSetValues` sets the resource value pair for a given widget using a `NULL` terminated list much like `XtVaCreateManagedWidget()` (*see below*).

Therefore, we can achieve the same result as above for setting the `button` in `push.c` via:

```
XtSetValues(button, XmNwidth, 500, XmNwidth, 750, NULL);
```

## Setting resources at creation

We can set resource values at creation. This is common if we wish to permanently override a default resource value. There are a couple of methods we can adopt to achieve this.

We can set an `Arg` argument list using `XtSetArg()` as in the previous section and then specify the `arg` list and the number of arguments in the `XmCreate...()` function. Therefore, we could amend the `push.c` program to set the resources at initialisation of the `button` widget by inserting the following code:

```
Arg args[2]; /* Arg array */
int n = 0; /* number arguments */
```

.....

```
XtSetArg(args[n], XmNwidth, 400);
n++;
XtSetArg(args[n], XmNwidth, 600);
n++;

button = XmCreatePushButton(top_wid, "Push_me", args, n);
```

There is an alternative method which lets us set resource values and create a managed widget in one function call. The function is `XtVaCreateManagedWidget()` and is a convenient way to program such tasks. Note that this is a *general* function that can create many different classes of widget. This function is preferred for the creation of widgets due to its uniformity of syntax/structure and its brevity in performing more than one task. Where appropriate all widgets will subsequently be created using this function.

Let us look at the syntax of this function:

```
Widget XtVaCreateManagedWidget(String name,
                                WidgetClass widget_class, Widget parent,
                                ... resource name/value pairs ...,
                                NULL)
```

where:

- name specifies the name for the created widget.
- widget\_class specifies the class of the widget.
- parent is the parent widget.
- There then follows a NULL terminated list of pairs of resource name and values.

The function returns a widget of the class specified.

Therefore, to create a managed PushButton widget with dimensions 400 by 300 we would type:

```
button = XtVaCreateManagedWidget("Push_me",
    XmPushButtonWidgetClass, top_wid,
    XmNwidth, 400,
    XmNheight, 200,
    NULL);
```

Here, `XmPushButtonWidgetClass` is the class identifier of a PushButton. Other widget types should be fairly obvious.

## Fallback Resources

Fallback resources are used as a mechanism where the specified resource value settings only take effect if all other resource setting methods have failed. Fallback resources are passed as arguments to the `XtVaAppInitialize()` function (Section 5.6.1). The fallback resources are passed as a NULL terminated list of strings to this function. Each string specifies a resource value setting similar to those developed for the user and class resource files. Therefore, to set fallback resources for the `push.c` program we could include the following code:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
```



```

/* Define fallback\_resources */

static String fallback\_resources[] = {
    "*width: 300",
    "*height: 400",
    NULL /* NULL termination */
};

main(int argc, char **argv)
{
    Widget      top_wid, button;
    XtAppContext app;

    top_wid = XtVaAppInitialize(&app,
                               "Push", /* class name */
                               NULL, 0, /* NO command line options table */
                               &argc, argv, /* command line arguments */
                               fallback\_resources, /* fallback\_resources list */
                               NULL);

    .....

```

## Exercises

NEED SOME

## Combining Widgets

The programs we have studied so far have used a top level (application) shell widget and a single child widget (e.g. PushButton).

Most Motif programs will need to employ a combination of many widgets, in order to produce an effective GUI. For example, a text editor application usually requires a combination of buttons, menus, text areas *etc.*

In this Chapter we will look at how we arrange widgets and furthermore explicitly position widgets within a GUI.

## Arranging and Positioning Widgets

We should now be familiar with the concept of a *tree of widgets* which is formed by creating widgets with other widgets as parents. When we combine widgets, we simply carry this principle further. Our major concern when combining widgets is to place them in some order and some relative position, with respect to other widgets. Usually we do not want widgets to obscure each other. Care must also be taken with the organisation and positioning of widgets when the window containing the widgets is resized. In particular:

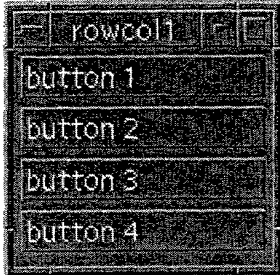
- Is it practical for the particular widget to be resized?
- Can the relative positioning between widgets be preserved as a result of the resize?

As we shall discover shortly, Motif provides a great deal of flexibility in the behaviour of widgets in such circumstances. The GUI programmer should carefully consider the means of interaction most suitable for the particular application and then select the most appropriate Motif widget and means of organisation to

achieve this.

The management of widget geometry is taken care of by certain **manager widgets**. The *RowColumn* and *Form* widgets are the most common widgets used for arranging widgets.

Consider a simple multiple widget program output (Fig 8.1).



**Fig. 8.1 Simple Multiple Widget Layout** This layout is usually specified by the widget tree structure illustrated in Figure 8.2.

The application shell is still be the top level, below this there would be a RowColumn or Form widget which would contain some primitive widgets (*e.g.* PushButtons) and define exactly how they are to be positioned relative to each other. Several possible arrangements are available and the format of each depends on the context of the application (*e.g.* how the widgets are displayed if the window size is increased or decreased).



**Fig. 8.2 Multiple Widget Tree** We will now consider how we can arrange widgets by considering the RowColumn and Form widgets.

## The RowColumn Widget

This the simplest widget in terms of how it manages the positioning of its child widgets. Widgets are positioned as follows:

- Consecutively created child widgets are layed out in a horizontal or vertical order depending on how the `XmNorientation` resource is set. `XmVERTICAL` is the default value, `XmHORIZONTAL` is the alternative value.
- To specify the number of rows/columns, set the `XmNnumcolumns` resource. This sets the number of columns if the `XmNorientation` is `XmVERTICAL`, otherwise it specifies the number of rows.
- Widgets must be the same size, otherwise the RowColumn widget will force widgets to be the same size. The resizing of widgets is controlled by the `XmNpacking` resource:
  - If the packing is set to `PACK_TIGHT` (default), then columns (rows if `XmHORIZONTAL` orientation) are forced to have the same width.
  - `PACK_COLUMN` makes all children the same size.
  - `PACK_NONE` disables any attempt to make the children regular in size.

Let us now look at two programs that illustrate the above principles by studying how we achieve the

outputs illustrated in Fig. 8.1 (rowcol1.c program) and Fig. 8.3 (rowcol2.c program). As can be seen in these figures, rowcol1.c lays 4 PushButtons vertically (default) whereas rowcol2.c sets the XmNorientation resource for a horizontal layout of the same 4 buttons. The output of the programs are illustrated in Fig 8.3.



**Fig. 8.3 RowColumn program output**

The rowcol1.c program is as follows:

```
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>

main(int argc, char **argv)
{
    Widget top_widget, rowcol;
    XtAppContext app;

    top_widget = XtVaAppInitialize(&app, "rowcol", NULL, 0,
        &argc, argv, NULL, NULL);

    rowcol = XtVaCreateManagedWidget("rowcolumn",
        xmRowColumnWidgetClass, top_widget, NULL);

    (void) XtVaCreateManagedWidget("button 1",
        xmPushButtonWidgetClass, rowcol, NULL);

    (void) XtVaCreateManagedWidget("button 2",
        xmPushButtonWidgetClass, rowcol, NULL);

    (void) XtVaCreateManagedWidget("button 3",
        xmPushButtonWidgetClass, rowcol, NULL);

    (void) XtVaCreateManagedWidget("button 4",
        xmPushButtonWidgetClass, rowcol, NULL);

    XtRealizeWidget(top_widget);
    XtAppMainLoop(app);
}
```

The rowcol1.c program does not really do much. It provides no callback functions to perform any tasks. It simply creates a RowColumn widget, rowcol, and creates 4 child buttons with this. Note that rowcol is a child of app -- the application shell widget. The <Xm/RowColumn.h> header file must also be included.

The rowcol2.c program is identical to rowcol1.c, except that the XmNorientation resource is set at the rowcol widget creation with:

```
rowcol = XtVaCreateManagedWidget("rowcolumn",
    xmRowColumnWidgetClass, top_widget,
    XmNorientation, XmHORIZONTAL,
    NULL);
```

## Forms

Forms are the other primary geometry manager widget. They allow more complex handling of positioning of child widgets and can handle widgets of different sizes.

There is more than one way to arrange widgets within a form. We will look at three programs `form1.c`, `form2.c` and `form3.c` that achieve similar results but illustrate different approaches to attaching widgets to forms. All three programs produce initial output illustrated in Fig. 8.4 however if the window is resized, the different attachment policies have a different effect (Figs 8.5, 8.6 and 8.7).



Fig. 8.4 `form1.c` initial output

## Simple Attachment -- `form1.c`

Widgets are placed in a form by specifying the attachment of widgets to *edges* of other widgets. The edges of widgets can either be on the form widget itself, or on other child widgets. Edges are referred to by top, bottom, left and right attachments. A widget has resources, such as `XmNtopattachment` to attach a widget, to an appropriate edge:

- To attach a widget to the parent form, set the resource value `XmATTACH_FORM`.
- To attach a widget to another widget, set the resource value `XmATTACH_WIDGET`.
- The widget that an attachment is made to must also be specified, by setting the resource `XmNtopWidget` to the appropriate widget.

Let us look at how all this works in the `form1.c` program:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>

main (int argc, char **argv)
{
    XtAppContext app;
    Widget      top_wid, form,
               button1, button2,
               button3, button4;
    int n=0;

    top_wid = XtVaAppInitialize(&app, "Form1",
                               NULL, 0, &argc, argv, NULL, NULL);

    /* create form and child buttons */

    form = XtVaCreateManagedWidget("form",
                                    xmFormWidgetClass, top_wid, NULL);

    button1 = XtVaCreateManagedWidget("Button 1",
                                       xmPushButtonWidgetClass, form,
                                       /* attach to top, left of form */
                                       XmNtopAttachment, XmATTACH_FORM,
                                       XmNleftAttachment, XmATTACH_FORM,
                                       NULL);
```

```

button2 = XtVaCreateManagedWidget("Button 2",
    xmPushButtonWidgetClass, form,
    XmNtopAttachment, XmATTACH_WIDGET,
    XmNtopWidget, button1, /* top to button 1 */
    XmNleftAttachment, XmATTACH_FORM, /* left, bottom to form */
    XmNbottomAttachment, XmATTACH_FORM,
    NULL);

button3 = XtVaCreateManagedWidget("Button 3",
    xmPushButtonWidgetClass, form,
    XmNtopAttachment, XmATTACH_FORM, /* top, right to form */
    XmNrightAttachment, XmATTACH_FORM,
    XmNleftAttachment, XmATTACH_WIDGET, /* left to button 1 */
    XmNleftWidget, button1,
    NULL);

button4 = XtVaCreateManagedWidget("Button 4",
    xmPushButtonWidgetClass, form,
    XmNbottomAttachment, XmATTACH_FORM, /* bottom right to form */
    XmNrightAttachment, XmATTACH_FORM,
    XmNtopAttachment, XmATTACH_WIDGET,
    XmNtopWidget, button3, /* top to button 3 */
    XmNleftAttachment, XmATTACH_WIDGET,
    XmNleftWidget, button2, /* left to button 2 */
    NULL);
XtRealizeWidget (top_wid);
XtAppMainLoop (app);
}

```

In the above program, the form widget is created as a child of app and 4 buttons are the children of form. The inclusion of `<Xm/Form.h>` header file is always required.

The attachment of the button widgets is as follows:

- button1 is simply attached to the top left of the form.
- button2 is attached to bottom left of the form and its top side is attached to button1.
- button3 is attached to the top right of the form and its right side is attached to button1.
- button4 is attached to the bottom right of the form and its top and left sides are attached to button3 and button2 respectively.

It is advisable to control the attachment as much as possible, so that any resizing of the form will still preserve the desired order. Fig 8.5 shows the effect of an enlargement of the window produce by the `form1.c` program. Notice that the relative sizes of individual buttons is not preserved.

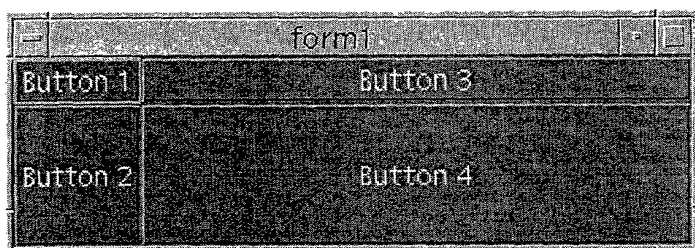


Fig. 8.4 `form1.c` resized window output

## Attach positions -- `form2.c`

We can position the side of a widget to a *position* in a form. Motif assumes that a form has been partitioned

into a number of segments. The position specified is the number of segments from the top left corner.

By default there is assumed to be 100 divisions along the top, bottom, left and right sides. This can be changed by setting the form resource

`XmNfractionBase`

The position of a particular side of a widget is set by the widget resource `XmNtopAttachment` *etc.* to `XmATTACH_POSITION` and then setting another resource `XmNtopPosition` *etc.* to the appropriate integer value.

The `form2.c` program specifies the top left of `button1` to the edges of the form. The right and bottom edges are attached half way (50 out of a 100 units) along the respective bottom and right edges of the *form*. The `button2`, `button3` and `button4` widgets are positioned similarly.

The complete `form2.c` program listing is:

```
#include <Xm/Xm.h>
#include <Xm/PushButton.h>
#include <Xm/Form.h>

main (int argc, char **argv)
{
    XtAppContext app;
    Widget      top_wid, form,
               button1, button2,
               button3, button4;
    int n=0;

    top_wid = XtVaAppInitialize(&app, "Form2",
                               NULL, 0, &argc, argv, NULL, NULL);

    /* create form and child buttons */

    form = XtVaCreateManagedWidget("form", xmFormWidgetClass,
                                    top_wid, NULL);

    button1 = XtVaCreateManagedWidget("Button 1",
                                       xmPushButtonWidgetClass, form,
                                       /* attach to top, left of form */
                                       XmNtopAttachment, XmATTACH_FORM,
                                       XmNleftAttachment, XmATTACH_FORM,
                                       XmNrightAttachment, XmATTACH_POSITION,
                                       XmNrightPosition, 50,
                                       XmNbottomAttachment, XmATTACH_POSITION,
                                       XmNbottomPosition, 50,
                                       NULL);

    button2 = XtVaCreateManagedWidget("Button 2",
                                       xmPushButtonWidgetClass, form,
                                       XmNbottomAttachment, XmATTACH_FORM,
                                       XmNleftAttachment, XmATTACH_FORM,
                                       XmNrightAttachment, XmATTACH_POSITION,
                                       XmNrightPosition, 50,
                                       XmNtopAttachment, XmATTACH_POSITION,
                                       XmNtopPosition, 50,
                                       NULL);

    button3 = XtVaCreateManagedWidget("Button 3",
```

```

xmPushButtonWidgetClass, form,
XmNtopAttachment, XmATTACH_FORM,
XmNrightAttachment, XmATTACH_FORM,
XmNleftAttachment, XmATTACH_POSITION,
XmNleftPosition, 50,
XmNbottomAttachment, XmATTACH_POSITION,
XmNbottomPosition, 50,
NULL);

```

```

button4 = XtVaCreateManagedWidget("Button 4",
xmPushButtonWidgetClass, form,
XmNbottomAttachment, XmATTACH_FORM,
XmNrightAttachment, XmATTACH_FORM,
XmNleftAttachment, XmATTACH_POSITION,
XmNleftPosition, 50,
XmNtopAttachment, XmATTACH_POSITION,
XmNtopPosition, 50,
NULL);

```

```

XtRealizeWidget (top_wid);
XtAppMainLoop (app);
}

```

One effect of this type of widget attachment is that the relative size of component widgets is preserved when the window containing these widgets is resized (Fig 8.6).

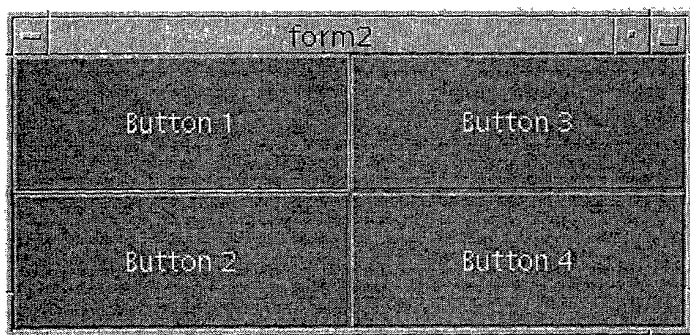


Fig. 8.6 form2.c output (resized)

## Opposite attachment -- form3.c

Yet another way to attach widgets is to place an edge opposite edges of another widget. This is achieved by setting the `XmNtopAttachment` *etc.* resources to `XmATTACH_OPPOSITE_WIDGET`. Just as with `XmATTACH_WIDGET`, a widget has to be associated with the `XmNwidget` resource.

With the opposite form of attachment "similar" edges are attached. In the `form3.c` program we attach the right edge of `button2` to the right edge of `button1`, the left edge of `button3` with the left of `button1` and so on.

The complete `form3.c` program listing is:

```

#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>

```

```

main (int argc, char **argv)
{
    XtAppContext app;
    Widget      top_wid, form,
                button1, button2,
                button3, button4;
    int n=0;

    top_wid = XtVaAppInitialize(&app, "Form3",
                                NULL, 0, &argc, argv, NULL, NULL);

    /* create form and child buttons */

    form = XtVaCreateManagedWidget("form", xmFormWidgetClass,
                                    top_wid, NULL);

    button1 = XtVaCreateManagedWidget("Button 1",
                                       xmPushButtonWidgetClass, form,
                                       /* attach to top, left of form */
                                       XmNtopAttachment, XmATTACH_FORM,
                                       XmNleftAttachment, XmATTACH_FORM,
                                       NULL);

    button2 = XtVaCreateManagedWidget("Button 2",
                                       xmPushButtonWidgetClass, form,
                                       XmNtopAttachment, XmATTACH_WIDGET,
                                       XmNtopWidget, button1,
                                       XmNleftAttachment, XmATTACH_FORM,
                                       XmNbottomAttachment, XmATTACH_FORM,
                                       XmNrightAttachment, XmATTACH_OPPOSITE_WIDGET,
                                       XmNrightWidget, button1,
                                       NULL);

    button3 = XtVaCreateManagedWidget("Button 3",
                                       xmPushButtonWidgetClass, form,
                                       XmNtopAttachment, XmATTACH_FORM,
                                       XmNrightAttachment, XmATTACH_FORM,
                                       XmNleftAttachment, XmATTACH_WIDGET,
                                       XmNleftWidget, button1,
                                       NULL);

    button4 = XtVaCreateManagedWidget("Button 4",
                                       xmPushButtonWidgetClass, form,
                                       XmNbottomAttachment, XmATTACH_FORM,
                                       XmNrightAttachment, XmATTACH_FORM,
                                       XmNtopAttachment, XmATTACH_WIDGET,
                                       XmNtopWidget, button3,
                                       XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET,
                                       XmNleftWidget, button3,
                                       NULL);

    XtRealizeWidget (top_wid);
    XtAppMainLoop (app);
}

```

The relative sizing of widgets within the window is not guaranteed to be preserved, as in `form1.c` (Fig 8.7). However this method of layout is sometimes a natural way to express the configuration of the widgets.



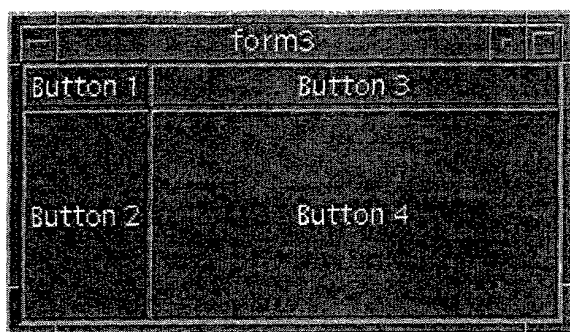


Fig. 8.7 form3.c output (resized)

## A more complete form program -- arrows.c

Let us finish off this section by building a Form widget that contains two different types of widget. It will also use callback functions thus making a more complete application program example.

The program is called `arrows.c`. It creates 4 ArrowButton widgets arranged in a north, south, east and west type arrangement. In the middle of the 4 ArrowButtons is a PushButton, labelled "Quit". The output of `arrows.c` is shown in Fig. 8.8.

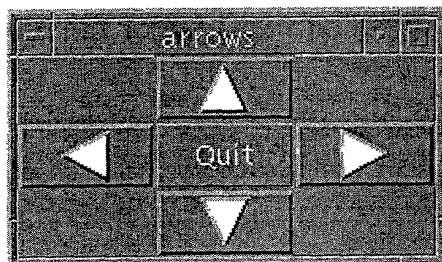


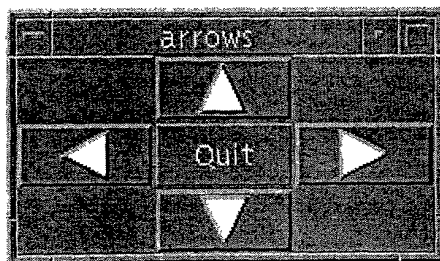
Fig. 8.8 arrows.c output

The program listing is:

## A more complete form program -- arrows.c

Let us finish off this section by building a Form widget that contains two different types of widget. It will also use callback functions thus making a more complete application program example.

The program is called `arrows.c`. It creates 4 ArrowButton widgets arranged in a north, south, east and west type arrangement. In the middle of the 4 ArrowButtons is a PushButton, labelled "Quit". The output of `arrows.c` is shown in Fig. 8.8.



**Fig. 8.8 arrows.c output**

The program listing is:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/ArrowB.h>
#include <Xm/Form.h>

/* Prototype callback fns */
void north(Widget , XtPointer ,
           XmPushButtonCallbackStruct *),
    south(Widget , XtPointer ,
          XmPushButtonCallbackStruct *),
    east(Widget , XtPointer ,
         XmPushButtonCallbackStruct *),
    west(Widget , XtPointer ,
         XmPushButtonCallbackStruct *),
    quitb(Widget , XtPointer ,
          XmPushButtonCallbackStruct *);

main(int argc, char **argv)
{
    XtAppContext app;
    Widget top_wid, form,
           arrow1, arrow2,
           arrow3, arrow4,
           quit;

    top_wid = XtVaAppInitialize(&app, "Multi Widgets",
                                NULL, 0, &argc, argv, NULL, NULL);

    form = XtVaCreateWidget("form", xmFormWidgetClass, top_wid,
                            XmNfractionBase, 3,
                            NULL);

    arrow1 = XtVaCreateManagedWidget("arrow1",
                                      xmArrowButtonWidgetClass, form,
                                      XmNtopAttachment, XmATTACH_POSITION,
                                      XmNtopPosition, 0,
                                      XmNbottomAttachment, XmATTACH_POSITION,
                                      XmNbottomPosition, 1,
                                      XmNleftAttachment, XmATTACH_POSITION,
                                      XmNleftPosition, 1,
                                      XmNrightAttachment, XmATTACH_POSITION,
                                      XmNrightPosition, 2,
                                      XmNarrowDirection, XmARROW_UP,
                                      NULL);

    arrow2 = XtVaCreateManagedWidget("arrow2",
                                      xmArrowButtonWidgetClass, form,
                                      XmNtopAttachment, XmATTACH_POSITION,
                                      XmNtopPosition, 1,
                                      XmNbottomAttachment, XmATTACH_POSITION,
                                      XmNbottomPosition, 2,
                                      XmNleftAttachment, XmATTACH_POSITION,
                                      XmNleftPosition, 0,
                                      XmNrightAttachment, XmATTACH_POSITION,
                                      XmNrightPosition, 1,
```

```

        XmNarrowDirection,    XmARROW_LEFT,
        NULL);

    arrow3 = XtVaCreateManagedWidget("arrow3",
        xmArrowButtonWidgetClass, form,
        XmNtopAttachment,     XmATTACH_POSITION,
        XmNtopPosition,       1,
        XmNbottomAttachment,   XmATTACH_POSITION,
        XmNbottomPosition,     2,
        XmNleftAttachment,     XmATTACH_POSITION,
        XmNleftPosition,       2,
        XmNrightAttachment,    XmATTACH_POSITION,
        XmNrightPosition,      3,
        XmNarrowDirection,     XmARROW_RIGHT,
        NULL);

    arrow4 = XtVaCreateManagedWidget("arrow4",
        xmArrowButtonWidgetClass, form,
        XmNtopAttachment,     XmATTACH_POSITION,
        XmNtopPosition,       2,
        XmNbottomAttachment,   XmATTACH_POSITION,
        XmNbottomPosition,     3,
        XmNleftAttachment,     XmATTACH_POSITION,
        XmNleftPosition,       1,
        XmNrightAttachment,    XmATTACH_POSITION,
        XmNrightPosition,      2,
        XmNarrowDirection,     XmARROW_DOWN,
        NULL);

    quit = XtVaCreateManagedWidget("Quit",
        xmPushButtonWidgetClass, form,
        XmNtopAttachment,     XmATTACH_POSITION,
        XmNtopPosition,       1,
        XmNbottomAttachment,   XmATTACH_POSITION,
        XmNbottomPosition,     2,
        XmNleftAttachment,     XmATTACH_POSITION,
        XmNleftPosition,       1,
        XmNrightAttachment,    XmATTACH_POSITION,
        XmNrightPosition,      2,
        NULL);

    /* add callback functions */

    XtAddCallback(arrow1, XmNactivateCallback, north, NULL);
    XtAddCallback(arrow2, XmNactivateCallback, west, NULL);
    XtAddCallback(arrow3, XmNactivateCallback, east, NULL);
    XtAddCallback(arrow4, XmNactivateCallback, south, NULL);
    XtAddCallback(quit, XmNactivateCallback, quitb, NULL);

    XtManageChild(form);
    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

```

```
/* CALLBACKS */
```

```
void north(Widget w, XtPointer client_data,
           XmPushButtonCallbackStruct *cbs)
```

```
{ printf("Going North\n");
}
```

```
void west(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)
```

```

    { printf("Going West\n");
    }

void east(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)

    { printf("Going East\n");
    }

void south(Widget w, XtPointer client_data,
           XmPushButtonCallbackStruct *cbs)

    { printf("Going South\n");
    }

void quitb(Widget w, XtPointer client_data,
           XmPushButtonCallbackStruct *cbs)

    { printf("quit button pressed\n");
      exit(0);
    }

```

The `arrows.c` program uses the *fraction base* positioning method of placing widgets within a form:

- The `XmNfractionBase` is reset to 3 thus creating a 3 by 3 grid for attaching widget edges.
- Each top, bottom, left and right edge of the 4 ArrowButtons and the "Quit" PushButton are attached to the appropriate position within the grid.
- Callback functions for each widget are added in the usual way. The 4 arrow widget callback functions simply print out their direction. The quit callback function exits the program.

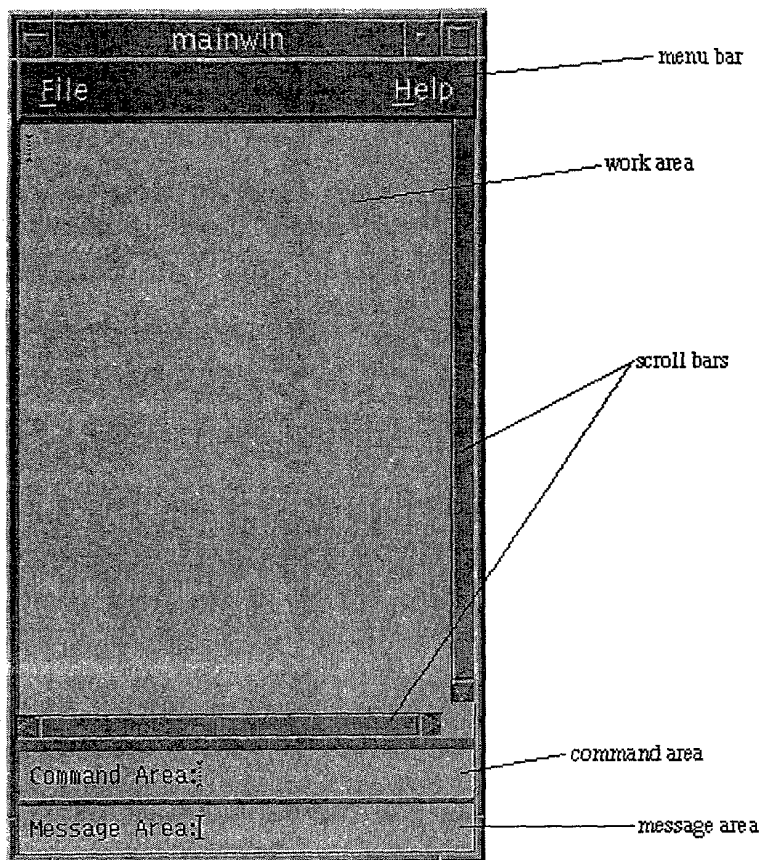
## Exercises

NEED SOME

## The Main Window Widget and Menus

When designing a GUI, care must be taken in the presentation of the application's primary window. This window is important as it is likely to be the major focus of the application -- the most visible and most used window in the application. In order to facilitate consistency amongst many different applications, Motif provides a general framework: the *Main Window Widget*, for an application developer to work with. The *Motif Style Guide* (Chapter 20) recommends that, whenever applicable, the Main Window window should be used. It should be noted, however, that the general framework of the Main Window is not always applicable to every application front end GUI. A text editor application front end is an example that might easily map into the Main Window framework, a simple calculator application most certainly would *not* fit the prescribed framework.

A Main Window widget can manage up to five specialised child widgets (Fig 9.1):



**Fig. 9.1 The MainWindow Widget with its Specialised Child Widgets**

- The *menu bar* -- to which a number of pull down menus can be attached.
- The *work area* -- the primary widget is placed here to perform the applications main functions.
- Vertical and Horizontal *scroll bars* -- for the work area.
- The *command area* -- where single-line commands can be entered by the user.
- The *message area* -- where the application can report back to the user.

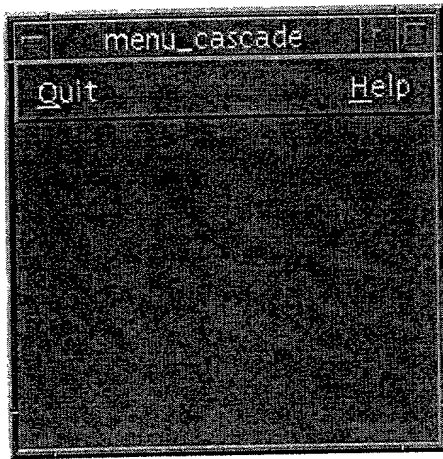
The MainWindow basically provides an efficient method of managing widgets as recommended by the *Motif Style Guide* (Chapter 20). In particular, the provision of a menu bar and work area is a convenient mechanism with which to drive many applications. It should be noted that the work area can be any widget (or composite hierarchy of widgets). The scrollbars, command and message area are optional.

In this Chapter, we will mainly concentrate our study on the relationship between the MainWindow and certain types of menus. We will also see how to put widgets in the work area MainWindow. We will see further applications of the MainWindow widget in the remainder of the book. The MainWindow will be used as a container widget in many example programs throughout the book.

## The MainWindow widget

As we have previously stated, the MainWindow is typically used as the top level container for an application's child widgets. The simplest functional MainWindow may consist of a menu bar along the top of the application and some work area below, this is illustrated in Fig. 9.2. We omit the scroll bars and

command and message areas for the time being.



**Fig. 9.2** menu\_cascade.c output

*Menu bar items* are placed within the menu bar . You can use menu bar items to perform selections directly. However, more usually a *PullDown* menu is attached to a menu bar item allowing greater selection opportunities.

The function of the work area is to perform the main application's functions. The work area can be any widget class. We will look at aspects of this in later sections when we have studied more widget classes.

Initially, we will concentrate on how to create menus in a MainWindow.

## The MenuBar

The creation of a fully functional pulldown MenuBar typical of most MainWindow based applications is fairly complex. We will, therefore, break it down into two stages.

1. We will create a simple MenuBar that lets us select actions directly from the bar (*MenuBar items*).
2. Having created simple MenuBar items we will attach pulldown menus to them.

### A simple MenuBar

A MenuBar widget is really a RowColumn widget under another name. The way we create a MenuBar is to use one of the (several) `XmCreate...` or `XtVaCreate...` Menu options. For most of our applications, we will use the `XmCreateMenuBar()` function.

Having created a MenuBar we create MenuBar items by attaching widgets to the MenuBar in exactly the same fashion as described for the RowColumn widget (Chapter 8). The widgets we usually attach are CascadeButton widgets since we can hang pull down menus off these widgets.

Fig. 9.2 shows the output of the menu\_cascade.c program that simply creates a simple MenuBar widget and attaches two CascadeButton widgets - help and quit. Minimal callback functions are associated with

each CascadeButton.

The full program listing of menu\_cascade.c is:

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>

/* Prototype callbacks */
void quit_call(void), help_call(void);

main(int argc, char **argv)
{
    Widget top_wid, main_w, menu_bar, quit, help;
    XtAppContext app;
    Arg arg[1];

    /* create application, main and menubar widgets */
    top_wid = XtVaAppInitialize(&app, "menu_cascade",
                               NULL, 0, &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
                                       xmMainWindowWidgetClass, top_wid,
                                       NULL);

    menu_bar = XmCreateMenuBar(main_w, "main_list",
                               NULL, 0);
    XtManageChild(menu_bar);

    /* create quit widget + callback */
    quit = XtVaCreateManagedWidget("Quit",
                                     xmCascadeButtonWidgetClass, menu_bar,
                                     XmNmnemonic, 'Q',
                                     NULL);

    XtAddCallback(quit, XmNactivateCallback, quit_call, NULL);

    /* create help widget + callback */
    help = XtVaCreateManagedWidget("Help",
                                     xmCascadeButtonWidgetClass, menu_bar,
                                     XmNmnemonic, 'H',
                                     NULL);

    XtAddCallback(help, XmNactivateCallback, help_call, NULL);

    /* Tell the menubar which button is the help menu */
    XtSetArg(arg[0], XmNmenuHelpWidget, help);
    XtSetValues(menu_bar, arg, 1);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}
```

```

void quit_call()
{
    printf("Quitting program\n");
    exit(0);
}

void help_call()
{
    printf("Sorry, I'm Not Much Help\n");
}

```

The first steps of the main function should now be familiar -- we initialise the application and create the `top_wid` top level widget.

A `MainWindow` widget, `main_win`, is then created as is a `MenuBar`, `menu_bar`. The `menu_bar` widget is a child of `main_win`. Finally, we attach two `CascadeButton` widgets, `quit` and `help`, as children of `menu_bar`. Note there is no work area created in this program.

A `CascadeButton` widget is usually used to attach a pulldown menu to the `MenuBar`. This `CascadeButton` widget is similar to the `PushButton` widget and can have callback functions attached to its `activateCallback` function -- illustrated in the `menu_cascade.c` program. However, it should be noted, that the main use of `CascadeButton` is to link a menu bar with a menu.

The program above simply attaches callback functions to the `CascadeButtons`. The callback functions do not do much except quit the program and print to standard output.

You can also associate a *mnemonic* to a particular menu selection. This means that you can use "hot keys" on the keyboard as a short cut to selection. You need to press `Meta` key plus the key in question.

In this program, we allow `Meta-Q` and `Meta-H` for the selection of `Quit` and `Help`. Note that Motif displays the appropriate `Meta` key by underlining the letter concerned on the menu bar (or menu item). The `XmNmnemonic` resource is used to select the appropriate keyboard short cut.

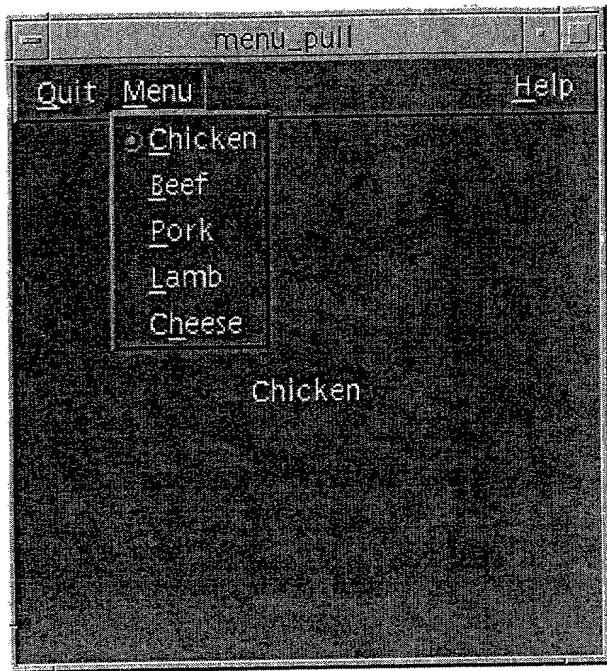
Apart from prescribing the use of the `Meta` key for the selection of menu items, the *Motif Style Guide* also insists that the `Help MenuBar` widget should always be placed on the right most side of the `MenuBar` (Fig. 9.2). This makes for easy location and selection of the help facility, should it exist.

The `MenuBar` resource, `XmNmenuHelpWidget`, is used to store the ID of the the appropriate widget (help in the above program).

## PullDown Menus

Let us now develop things a little further by adding pulldown menus to our `MenuBar`. A pulldown menu looks like this:





**Fig. 9.3** menu\_pull.c output

In order to see how we create and use pulldown menu items we will develop a program, menu\_pull.c that will create two pulldown menus:

- Quit which will contain a single item that lets us terminate our program.
- Menu which has 5 options that change the XmNlabelString of a label widget attached to the MainWindow as the work area.

We also attach the Help Cascade button as in the previous menu\_cascade.c program.

We should now be familiar with the first few steps of this program: We create the top level widget hierarchy as usual, with the MainWindow widget being a child of the application and a MenuBar widget a child of the MainWindow.

The complete listing of menu\_pull.c is as follows:

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/Label.h>

/* prototype functions */

void quit_call(Widget , int),
    menu_call(Widget , int),
    help_call(void);

Widget label;
String food[] = { "Chicken", "Beef", "Pork", "Lamb", "Cheese"};
```

```

main(int argc, char **argv)

{
    Widget top_wid, main_w, help;
    Widget menubar, menu, widget;
    XtAppContext app;
    XColor back, fore, spare;
    XmString quit, menu_str, help_str, chicken, beef, pork,
        lamb, cheese, label_str;

    int n = 0;
    Arg args[2];

    /* Initialize toolkit */
    top_wid = XtVaAppInitialize(&app, "Demos",
        NULL, 0, &argc, argv, NULL, NULL);

    /* main window will contain a MenuBar and a Label */
    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass, top_wid,
        XmNwidth, 300,
        XmNheight, 300,
        NULL);

    /* Create a simple MenuBar that contains three menus */
    quit = XmStringCreateLocalized("Quit");
    menu_str = XmStringCreateLocalized("Menu");
    help_str = XmStringCreateLocalized("Help");

    menubar = XmVaCreateSimpleMenuBar(main_w, "menubar",
        XmVaCASCADEBUTTON, quit, 'Q',
        XmVaCASCADEBUTTON, menu_str, 'M',
        XmVaCASCADEBUTTON, help_str, 'H',
        NULL);

    XmStringFree(menu_str); /* finished with this so free */
    XmStringFree(help_str);

    /* First menu is the quit menu -- callback is quit_call() */
    XmVaCreateSimplePulldownMenu(menubar, "quit_menu", 0, quit_call,
        XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
        NULL);
    XmStringFree(quit);

    /* Second menu is the food menu -- callback is menu_call() */
    chicken = XmStringCreateLocalized(food[0]);
    beef = XmStringCreateLocalized(food[1]);
    pork = XmStringCreateLocalized(food[2]);
    lamb = XmStringCreateLocalized(food[3]);
    cheese = XmStringCreateLocalized(food[4]);

    menu = XmVaCreateSimplePulldownMenu(menubar, "edit_menu", 1,
        menu_call,
        XmVaRADIOBUTTON, chicken, 'C', NULL, NULL,
        XmVaRADIOBUTTON, beef, 'B', NULL, NULL,
        XmVaRADIOBUTTON, pork, 'P', NULL, NULL,
        XmVaRADIOBUTTON, lamb, 'L', NULL, NULL,
        XmVaRADIOBUTTON, cheese, 'h', NULL, NULL,
        /* RowColumn resources to enforce */
        XmNradioBehavior, True,
        /* select radio behavior in Menu */

```

```

        XmNradioAlwaysOne, True,
        NULL);
XmStringFree(chicken);
XmStringFree(beef);
XmStringFree(pork);
XmStringFree(lamb);
XmStringFree(cheese);

/* Initialize menu so that "chicken" is selected. */
if (widget = XtNameToWidget(menu, "button_1"))
{
    XtSetArg(args[n],XmNset, True);
    n++;
    XtSetValues(widget, args, n);
}

n=0; /* reset n */

/* get help widget ID to add callback */

help = XtVaCreateManagedWidget( "Help",
    xmCascadeButtonWidgetClass, menubar,
    XmNmnemonic, 'H',
    NULL);

XtAddCallback(help, XmNactivateCallback, help_call, NULL);

/* Tell the menubar which button is the help menu */

XtSetArg(args[n],XmNmenuHelpWidget,help);
n++;
XtSetValues(menubar,args,n);
n=0; /* reset n */

XtManageChild(menubar);

/* create a label text widget that will be "work area"
   selections from "Menu" menu change label
   default label is item 0 */

label_str = XmStringCreateLocalized(food[0]);

label = XtVaCreateManagedWidget("main_window",
    xmLabelWidgetClass, main_w,
    XmNlabelString, label_str,
    NULL);

XmStringFree(label_str);

/* set the label as the "work area" of the main window */
XtVaSetValues(main_w,
    XmNmenuBar, menubar,
    XmNworkWindow, label,
    NULL);

XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

/* Any item the user selects from the File menu calls this function.
   It will "Quit" (item_no == 0). */

```

```

void
quit_call(Widget w, int item_no)

    /* w = menu item that was selected
       item_no = the index into the menu */

{

    if (item_no == 0) /* the "quit" item */
        exit(0);

}

/* Called from any of the food "Menu" items.
   Change the XmNlabelString of the label widget.
   Note: we have to use dynamic setting with setargs().
   */

void menu_call(Widget w, int item_no)

{
    int n = 0;
    Arg args[1];

    XmString    label_str;

    label_str = XmStringCreateLocalized(food[item_no]);


    XtSetArg(args[n], XmNlabelString, label_str);
    ++n;
    XtSetValues(label, args, n);
}

void help_call()
{
    printf("Sorry, I'm Not Much Help\n");
}

```

In this program we create the MenuBar widget with the convenience function `XmVaCreateSimpleMenuBar()` :

- The first two arguments of this function specify the *parent widget* (The Main Window, `main_w`, in this case) and the *name* of the widget, respectively.
- The following arguments are a NULL terminated variable length list of resource name/value pairs.

In `menu_pull.c`, we set up two CascadeButtons  by setting the `XmVACASCADEBUTTON` resource. Two arguments are associated with this resource:

- A *label* specified by an `XmString`.
- The *mnemonic* or *Meta* key associated with the `CascadeButton`.

Note: we convert a String to an `XmString` with the `XmStringCreateLocalized()` function.

It is good practice to free an `XmString` as soon as you have finished using it with the `XmStringFree()` command. Several "open" `XmStrings` can occupy a significant amount of application memory.

The creation of a pulldown menu is now relatively straightforward:

- We create a `PullDownMenu` widget
- Attach the `PullDownMenu` widget to a `MenuBar` item by simply making the `PullDownMenu` Widget a child of the `MenuBar`'s `CascadeButton`.

Let us now look at how we create the `Quit` menu:

```
quit_w = XmVaCreateSimplePulldownMenu(menubar, "quit_menu",
                                       0, quit_call,
                                       XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
                                       NULL);
```

We have used the Motif convenience function `XmVaCreateSimplePulldownMenu()` to return a `PulldownMenu` widget. This function has several arguments:

- The first argument is the parent widget (`MenuBar` in this example).
- The second is the name given to the widget for resource lookup.
- The third argument is the integer *ID* that specifies which `CascadeButton` (from the `MenuBar`) the `PulldownMenu` is attached to.

Thus, in this program an integer *ID* of 0 is attached to the `Quit` button and an *ID* of 1 would be attached to the `Menu` button.

- The fourth argument specifies the callback function associated with a menu choice.

**Note:** We do not specify a callback with an `XtAddCallback()` call in this instance.

- A `NULL` terminated resource name/value list. The list specifies the type and values of `PulldownMenu` items. Many possible classes are allowed, including further `CascadeButtons`, `RadioButtons`, `CheckButtons`. This program creates `PushButton` Menu items:

To specify a `PushButton` menu item, set a `XmVaPUSHBUTTON` resource list item and corresponding `XmString` label and Meta key accordingly for the list entry. `XmVaPUSHBUTTON` actually takes four arguments, the last two are only needed for advanced Motif use, so are not considered here -- they are just set to `NULL`.

The `Menu` menu is created in a similar way except that we have 5 menu items.

We may have one, minor, problem when assigning Meta keys. This is illustrated for the `Menu` items since we cannot have the same Meta key for two menu selections. So `Meta-B` is chosen for Beef and `Meta-L` for Lamb, *etc*. However, `Chicken` and `Cheese` must be assigned different Meta keys, so we allocate `Meta-C` for Chicken and `Meta-h` for cheese selections.

The last thing we need to look at is how we find out which selection has been made in our program.

Each `PulldownMenu` has an associated callback function. The callback function of a pulldown has two parameters, which we must define.

- The widget that called the function.
- An index to the menu item selected (starts at 0).

So, in the Quit callback, `quit_call()`, we only have one possible selection (`item_no` must equal zero).

In the Menu callback, `menu_call()`, the index corresponds to a food item setting of Chicken, Beef, ... etc..

Chapter 20 discusses aspects of the *Motif Style* guidelines for menus which also incorporate some general menu design issues.

## Tear-off menus

*Tear-off menus* allow the user to remove (or *tear*) a menu off the MenuBar and keep it displayed in a small dialog window (Figs. 9.4 and 9.5) on the screen until the user closes it from the window menu. The *Motif Style Guide* (Chapter 20) prescribes this for menus that are frequently used in order to ease menu selection. In order to make a menu a tear-off variety the `XmNtearOffModel` resource for a `PullDownMenu` widget needs to set to `XmTEAR_OFF_ENABLED`. If a menu is `XmTEAR_OFF_ENABLED` then its appearance is modified to include a small perforated line at the top of the menu (Fig. 9.4).



Fig. 9.4 A Tear-off Menu on the MenuBar

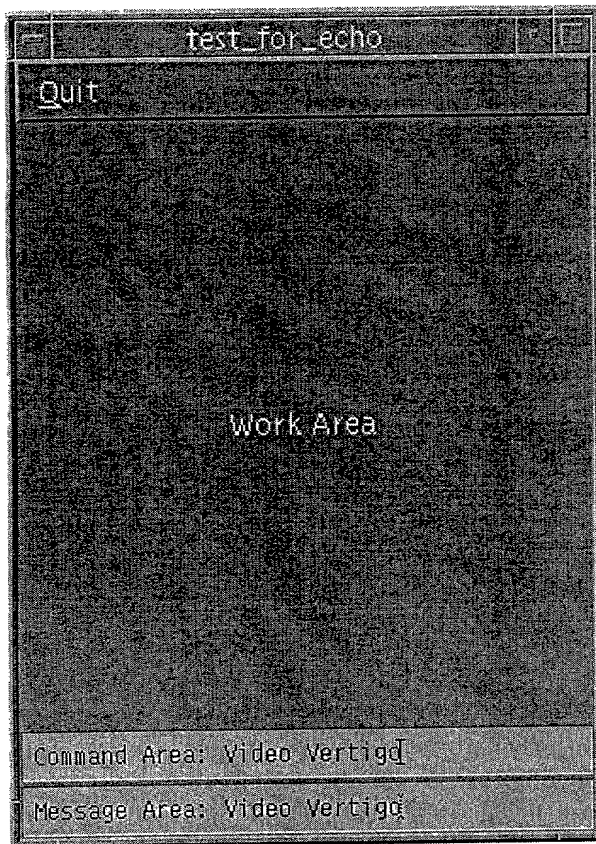


Fig. 9.5 A Tear-off Menu Dialog

## Other Main Window children

So far in this Chapter we have concentrated on the MenuBar and work area parts of the Main Window. In this section we will develop a simple program, `test_for_echo.c`, which creates and uses the command and message areas of the Main Window. We create a minimal MenuBar that simply allows you to quit the program. The work area created is a Label widget that does not perform any useful task in this example. The command and message areas created are both *TextField* widgets (see Chapter 11 for a complete description of the *TextField* widget). The command area receives (String) input and echoes it to the message area (Fig 9.6). The message area is not usually allowed to receive any user input. In this example we set the `XmNeditable` resource for the message area to be `False`.

*TextField*, *Label* or *Command* widgets are typically employed as the command and message areas.



**Fig. 9.6** The `test_for_echo.c` program output The `test_for_echo.c` program achieves this by:

- Creating a Menubar, Label and 2 TextField widgets and assigning their IDs to the MainWindow resources `XmNmenuBar`, `XmNworkWindow`, `XmNcommandWindow` and `XmNmessageWindow` respectively to form the complete MainWindow widget (Fig 9.6).
- A callback, `cmd_cbk`, is attached to the `XmNactivateCallback` event for the command area widget.
- The `cmd_cbk` callback parses the input command area widget, `cmd_widget`, for the input string and replaces the message area widget, `msg_wid`, with this string.
- Section 9.3 in the following Chapter explains the text handling part of this program further.

The complete program listing for `test_for_echo.c` is as follows:

```
#include <Xm/MainW.h>
#include <Xm/Label.h>
#include <Xm/Command.h>
#include <Xm/TextF.h>

#include <stdio.h>
#include <string.h> /* For String Handling */

#define MAX_STR_LEN 30 /* Max Char length of Text Field */

/* Callback function prototypes */

void cmd_cbk(), quit_cbk();

Widget msg_wid;

char *cmd_label = "Command Area: ";
char *msg_label = "Message Area: ";
```

```

int cmd_label_length;
int msg_label_length;

main(int argc, char **argv)
{
    Widget          top_wid, main_win, menubar, menu,
                    label, cmd_wid;
    XtAppContext    app;
    XmString        label_str, quit;

    cmd_label_length = strlen(cmd_label);
    msg_label_length = strlen(msg_label);

    /* initialize toolkit and create top_widlevel shell */
    top_wid = XtVaAppInitialize(&app, "Main Window",
                                NULL, 0, &argc, argv, NULL, NULL);

    /* Create MainWindow */

    main_win = XtVaCreateWidget("main_w",
                                xmMainWindowWidgetClass, top_wid,
                                XmNcommandWindowLocation, XmCOMMAND_BELOW_WORKSPACE,
                                NULL);

    /* Create a simple MenuBar that contains one menu */
    quit = XmStringCreateLocalized("Quit");
    menubar = XmVaCreateSimpleMenuBar(main_win, "menubar",
                                       XmVaCASCADEBUTTON, quit, 'Q',
                                       NULL);

    menu = XmVaCreateSimplePulldownMenu(menubar, "file_menu", 0,
                                       quit_cbk,
                                       XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
                                       NULL);

    XmStringFree(quit);

    /* Manage Menubar */

    XtManageChild(menubar);

    /* create a label text widget that will be work area */
    label_str = XmStringCreateLocalized("Work Area");

    label = XtVaCreateManagedWidget("main_window",
                                     xmLabelWidgetClass, main_win,
                                     XmNlabelString, label_str,
                                     XmNwidth, 1000,
                                     XmNheight, 800,
                                     NULL);

    XmStringFree(label_str);

    /* Create the command area */

    cmd_wid = XtVaCreateWidget("Command",
                               xmTextFieldWidgetClass, main_win,
                               XmNmaxLength, MAX_STR_LEN,
                               NULL);

```



```

XmTextSetString(cmd_wid,cmd_label);
XmTextSetInsertionPosition(cmd_wid, cmd_label_length);

XtAddCallback(cmd_wid, XmNactivateCallback, cmd_cbk);

XtManageChild(cmd_wid);

/* Create the message area */

msg_wid= XtVaCreateWidget( "Message:",
    xmTextFieldWidgetClass, main_win,
    XmNeditable, False,
    XmNmaxLength, MAX_STR_LEN,
    NULL);

XmTextSetString(msg_wid,msg_label);

XtManageChild(msg_wid);

/* set the label as the work, command and message areas
of the main window */

XtVaSetValues(main_win,
    XmNmenuBar, menubar,
    XmNworkWindow, label,
    XmNcommandWindow, cmd_wid,
    XmNmessageWindow,msg_wid,
    NULL);

XtManageChild(main_win);
XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

/* execute the command and redirect message area */

void cmd_cbk(Widget cmd_widget, XtPointer *client_data,
    XmAnyCallbackStruct *cbs)
{
    char cmd[MAX_STR_LEN],msg[MAX_STR_LEN];

    XmTextGetSubstring(cmd_widget,cmd_label_length,
        MAX_STR_LEN - cmd_label_length, MAX_STR_LEN ,cmd);

    /* Append input message to Message area */

    XmTextReplace(msg_wid,msg_label_length, MAX_STR_LEN,cmd);

    /* Reset Command Area label and insertion point*/

    XmTextSetString(cmd_widget, cmd_label);
    XmTextSetInsertionPosition(cmd_widget, cmd_label_length);
}

void quit_cbk(Widget w, int item_no)
{
    if (item_no == 0) /* the "quit" item */
        exit(0);
}

```

## Exercises

NEEDS SOME

## Dialog Widgets

Any application needs to interact with the user. At the simplest level an application may need to inform, alert or warn the user about its current state. More advanced interaction may require the user to select or input data. Selecting files from a directory/file selection window is typical of an advanced example. Clearly, the provision of such interaction is the concern of the GUI. Motif provides a variety of *Dialog* widgets or *Dialogs* that facilitate most common user interaction requirements.

## What are Dialogs?

Motif Dialog widgets usually comprise of the following components:

- A *dialog box* -- Dialogs put up a box with a message in and may also prompt the user for some input.
- Three buttons may also be provided:
  - **Ok** and **Cancel**, used, perhaps, to acknowledge the message and remove the Dialog box. The Dialog usually remains on the screen until either the **Ok** or **Cancel** button has been pressed. The **Return** key can also be used to acknowledge the Dialog.
  - **Help**, some additional information may be provided by pressing this button.
- You have probably seen this in action with a text editor putting up a message of the form ``Cannot open file ....''

More advanced Dialogs (*e.g. selection dialogs*) may significantly enhance this model.

Dialogs have many distinct uses, indeed the *Motif Style Guide* [Ope93] (Chapter 20) is specific in the use of each Dialog . The following Dialogs are provided by Motif:

### WarningDialog

-- Warns User about a program mishap.

### ErrorDialog

-- Alerts User to errors in the program.

### InformationDialog

-- Program Information supplied.

### PromptDialog

-- Allows user to supply data to program.

### QuestionDialog

-- Yes/No type queries.

### WorkingDialog

-- Notifies user if program is busy.

### SelectionDialog

-- Selection from a list of options.

### FileSelectionDialog

-- Specialised Selection Dialog to select directories and files.

### BulletinBoardDialog

-- Allows customised Dialogs to be created.

# Basic Dialog Management

To create a Dialog use one of the `XmCreate.....Dialog()` functions.

Dialogs do not usually appear immediately on screen after creation or when the the initial application GUI is realized. Indeed, if an application runs successfully certain Dialog widgets (Error or Warning Dialogs, in particular) may never be required. However, prudent applications should consider all practical avenues that an application would be expected to take and provide suitable information (via Dialogs) to the user.

It is advisable to create all Dialogs when the application is initialised and the overall GUI is setup. However, Dialogs will not be managed initially. Recall Section [5.7](#)) when a widget is *unmanaged* by its parent it will always be invisible.

Therefore, Dialogs are usually created *unmanaged* and displayed when required in the program as described below:

- To make a Dialog appear in your program you must *manage* the widget, explicitly. `XtManageChild()` is the function typically employed.
- To make one disappear you must *unmanage* it, explicitly. `XtUnmanageChild()` is the function typically employed.

This method has the advantage that we only need to create a Dialog once and can then manage or unmanage it when necessary.

Most Motif Dialogs have default callback resources attached to the common **Ok** and **Cancel**. One consequence of these default callbacks is that they will *unmanage* the widgets from which they were called. However, if the application provides alternative callback (or other) functions then responsibility for correctly managing and unmanaging them is given over to the programmer.

The use of many dialogs is very similar. We will study a few specific Dialogs in detail.

## The WarningDialog

This dialog is used to inform the user of a possible mistake in the program or in interaction with the program.

A typical example might be when you select the quit button (or menu item) to terminate the program -- if this was selected mistakenly, and there is no warning prompt, you have problems.

The `dialog1.c` (Section [10.5](#)) program attaches a pop-up WarningDialog when the quit menu option is selected (Fig [10.1](#)). If you now select **OK** the program terminates, **Cancel** returns back to the program.

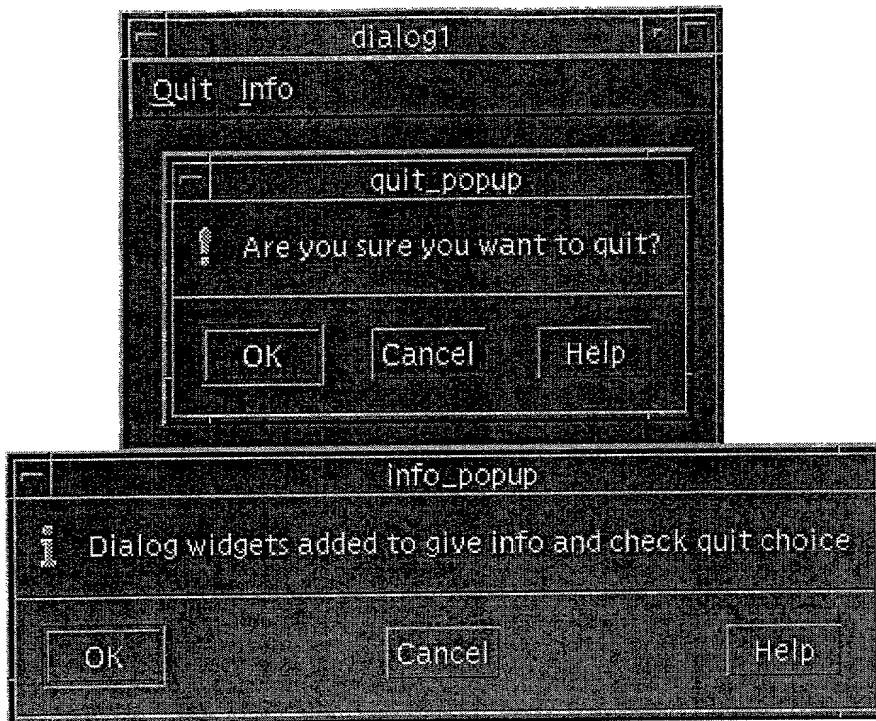


Fig. 10.1 WarningDialog and InformationDialog Widgets

To Create a WarningDialog:

The function `create_dialogs()` in `dialog1.c` creates the WarningDialog in the following typical manner:

- All the warning Dialog contains is an `xmString`, which is the prompt to the user.
  - For the Quit button we ask: "Are you sure you want to quit?".
  - So, we set the `xm_string` variable accordingly.
  - Set the `xmNmessageString` resource to the `xm_string` value.
  - Create the WarningDialog widget with the `xmCreateWarningDialog()` function.
  - Add callback functions to **Ok**, **Cancel** or **Help** buttons of Dialog.
  - In this case, we only need to attach an application specific callback to the **Ok** button. The resource `xmNokCallback` therefore needs a function attached. Other button presses will use default callbacks.
- NOTE:** The **Help** button does not do anything useful yet.

The callback function `quit_pop_up()` simply needs to `XtManageChild()` the given Dialog widget so that it is displayed as required by the application.

## The InformationDialog Widget

The InformationDialog Widget is essentially the same as the WarningDialog, except that InformationDialogs are intended to supply program information or help. In terms of Motif creation and management the widgets are identical except that the `xmCreateInformationDialog()` is used to create

this class of Dialog. The main difference between the widgets to the user is visual. Motif employs different icons to distinguish between the two (Fig 10.1).

The program `dialog1.c` illustrates the creation and use of an `InformationDialog` Widget.

## The `dialog1.c` program

This program uses Warning and Information Dialogs. These Dialogs are based on the `MessageBox` widget and so we must include `<Xm/MessageB.h>` header file.

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>

/* Prototype functions */

void create_dialogs(void);

/* callback for the pushbuttons. pops up dialog */
void info_pop_up(Widget , char *, XmPushButtonCallbackStruct *),
quit_pop_up(Widget , char *, XmPushButtonCallbackStruct *);

void info_activate(Widget ),
quit_activate(Widget );

/* Global reference for dialog widgets */
Widget info, quit;
Widget info_dialog, quit_dialog;

main(int argc, char *argv[])
{
    XtAppContext app;
    Widget top_wid, main_w, menu_bar;

    top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
        &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass, top_wid,
        XmNheight, 300,
        XmNwidth, 300,
        NULL);

    menu_bar = (Widget) XmCreateMenuBar(main_w, "main_list", NULL, 0);
    XtManageChild(menu_bar);

    /* create quit widget + callback */

    quit = XtVaCreateManagedWidget("Quit",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'Q',
        NULL);

    /* Callback has data passed to */
```

```

XtAddCallback(quit, XmNactivateCallback, quit_pop_up, NULL);

/* create help widget + callback */

info = XtVaCreateManagedWidget( "Info",
    xmCascadeButtonWidgetClass, menu_bar,
    XmNMnemonic, 'I',
    NULL);

XtAddCallback(info, XmNactivateCallback, info_pop_up, NULL);

/* Create but do not show (manage) dialogs */

create_dialogs();

XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

void create_dialogs()
{
    /* Create but do not manage dialog widgets */

    XmString xm_string;
    Arg args[1];

    /* Create InformationDialog */

    /* Label the dialog */

    xm_string = XmStringCreateLocalized("Dialog widgets added to \
        give info and check quit choice");
    XtSetArg(args[0], XmNmessageString, xm_string);

    /* Create the InformationDialog */

    info_dialog = XmCreateInformationDialog(info, "info", args, 1);
    XmStringFree(xm_string);

    XtAddCallback(info_dialog, XmNokCallback, info_activate, NULL);

    /* Create Warning Dialog */

    /* label the dialog */

    xm_string =
        XmStringCreateLocalized("Are you sure you want to quit?");
    XtSetArg(args[0], XmNmessageString, xm_string);

    /* Create the WarningDialog */

    quit_dialog = XmCreateWarningDialog(quit, "quit", args, 1);

    XmStringFree(xm_string);

    XtAddCallback(quit_dialog, XmNokCallback, quit_activate, NULL);
}

```

```

void info_pop_up(Widget cascade_button, char *text,
                 XmPushButtonCallbackStruct *cbs)
{
    XtManageChild(info_dialog);
}

void quit_pop_up(Widget cascade_button, char *text,
                 XmPushButtonCallbackStruct *cbs)
{
    XtManageChild(quit_dialog);
}

/* callback routines for dialogs */

void info_activate(Widget dialog)
{
    printf("Info Ok was pressed.\n");
}

void quit_activate(Widget dialog)
{
    printf("Quit Ok was pressed.\n");
    exit(0);
}

```

## Error, Working and Question Dialogs

These 3 Dialogs are similar to both the Information and Warning Dialogs. They are created and used in similar fashions. The main difference again being the icon used to depict the Dialog class as illustrated in Figs. 10.2 -- 10.4.



Fig.  ErrorDialog Widget

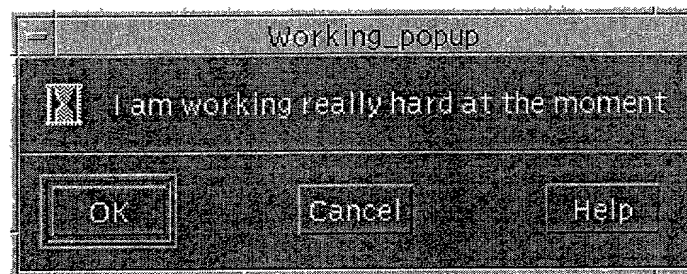


Fig.  WorkingDialog Widget

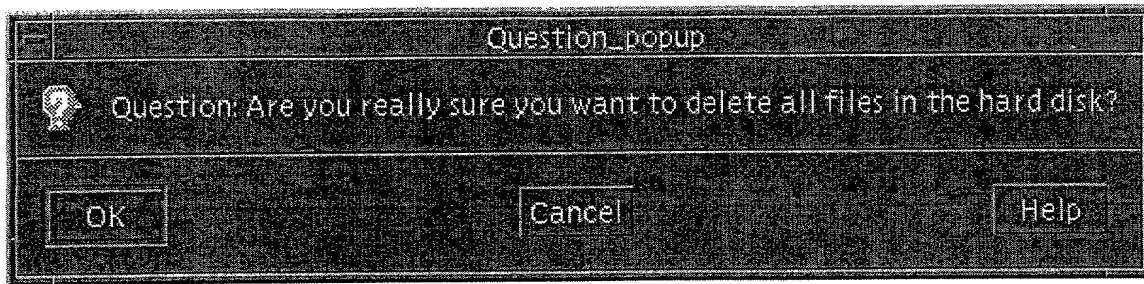


Fig.  QuestionDialog Widget

## Unwanted Dialog Buttons

When you create a Dialog, Motif will create 3 buttons by default -- **Ok**, **Cancel** and **Help**. There are many occasions when it is not natural to require the use of three buttons within an application. For instance in `dialog1.c` we only really need the user to acknowledge the InformationDialog and no user should need any help to choose whether to quit our program.

Motif provides a mechanism to disable unwanted buttons in a Dialog.

To remove a button:

- Use `XmMessageBoxGetChild()` (or similar function) to get the button widget's ID for the given dialog widget and button type.
- `XmDIALOG_OK_BUTTON`, `XmDIALOG_CANCEL_BUTTON` or `XmDIALOG_HELP_BUTTON` specify the type of the button.
- Unmanage the button widget.

An example where we disable the **Cancel** and **Help** buttons on the InformationDialog and the **Help** button on the WarningDialog from the Dialogs created in program `dialog1.c` is shown in Fig. [10.5](#). The code that performs the task of deleting the **Help** from a widget, `dialog` is as follows:

```
Widget remove;

remove = XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON);

XtUnmanageChild(remove);
```



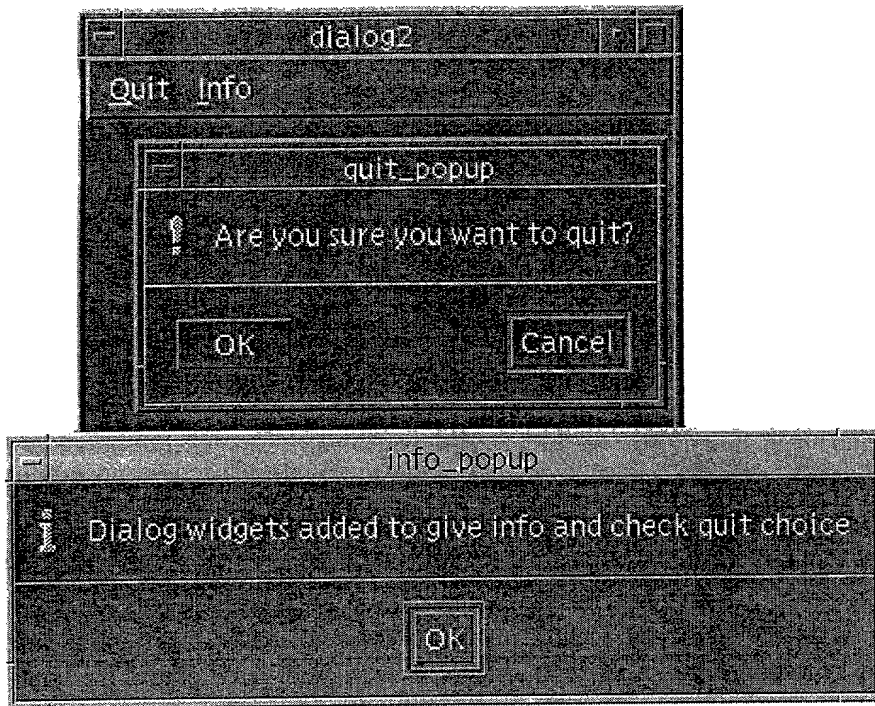


Fig. 10.5 Removed Dialog Button

## The PromptDialog Widget

The PromptDialog widget is slightly more advanced than the classes of Dialog widgets encountered so far. This widget allows the user to enter text (Fig. 10.6).

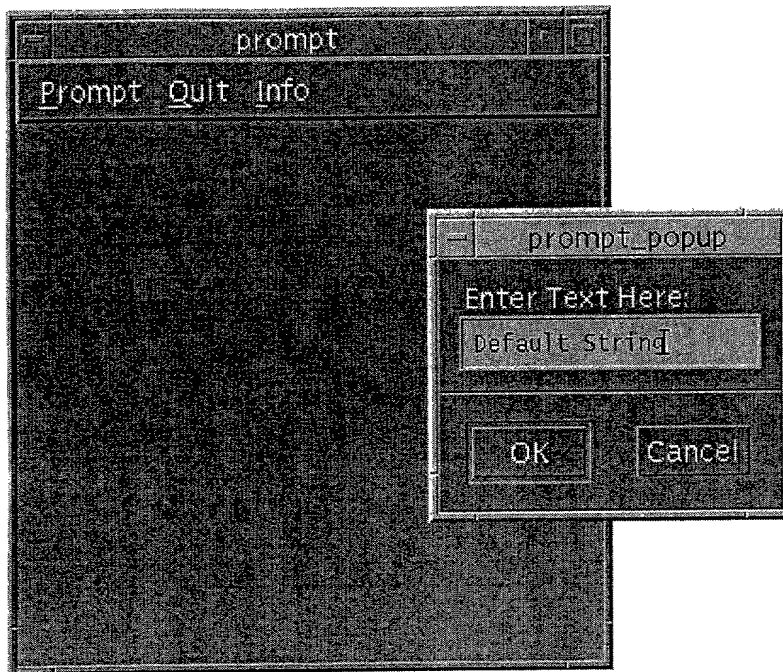
The function `XmCreatePromptDialog()` instantiates the Dialog. Typically two resources of a PromptDialog widget are required to be set. These resources are

```
XmNselectionLabelString
    -- the prompt message, an XmString data type.
XmNtextString
    -- the default response XmString which may be empty.
```

Note: A Prompt Dialog is based on the SelectionBox widget and so we must include `<Xm/SelectionBox.h>` header file.

### The prompt.c program

This program, an extension of `dialog1.c`, creates a PromptDialog into which the user can enter text. The PromptDialog first displayed by this program is shown in Fig. 10.6. The text is echoed in an InformationDialog which is created in a Prompt callback function, `prompt_activate()`.



**Fig. 10.6 PromptDialog Widget**

A PromptDialog Callback has the following structure

```
void prompt_callback(Widget widget, XtPointer client_data,
XmSelectionBoxCallbackStruct *selection)
```

Normally, we will only be interested in obtaining the string entered to the PromptDialog. An element of the XmSelectionBoxCallbackStruct, **value** holds the (XmString data type) value.

In prompt.c, the callback for the prompt dialog (activated with **Ok** button) -- prompt\_activate().

This function uses the selection->value XmString to set up the InformationDialog message String.

Note, that since a PromptDialog is a SelectionBox widget type we must use XmSelectionBoxGetChild() to find any buttons we may wish to remove from the PromptDialog. In prompt.c we remove the **Help** button in this way.

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/SelectioB.h>

/* Callback and other function prototypes */

void ScrubDial(Widget, int);
void info_pop_up(Widget, char *, XmPushButtonCallbackStruct *),
quit_pop_up(Widget, char *, XmPushButtonCallbackStruct *),
prompt_pop_up(Widget, char *, XmPushButtonCallbackStruct *);
void prompt_activate(Widget, caddr_t,
XmSelectionBoxCallbackStruct *);
void quit_activate(Widget);
```

```

Widget top_wid;

main(int argc, char *argv[])
{
    XtAppContext app;
    Widget main_w, menu_bar, info, prompt, quit;

    top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
                               &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
                                       xmMainWindowWidgetClass, top_wid,
                                       XmNheight, 300,
                                       XmNwidth, 300,
                                       NULL);

    menu_bar = XmCreateMenuBar(main_w, "main_list",
                               NULL, 0);
    XtManageChild(menu_bar);

    /* create prompt widget + callback */
    prompt = XtVaCreateManagedWidget("Prompt",
                                       xmCascadeButtonWidgetClass, menu_bar,
                                       XmNmnemonic, 'P',
                                       NULL);

    /* Callback has data passed to */
    XtAddCallback(prompt, XmNactivateCallback,
                  prompt_pop_up, NULL);

    /* create quit widget + callback */

    quit = XtVaCreateManagedWidget("Quit",
                                     xmCascadeButtonWidgetClass, menu_bar,
                                     XmNmnemonic, 'Q',
                                     NULL);

    /* Callback has data passed to */

    XtAddCallback(quit, XmNactivateCallback, quit_pop_up,
                  "Are you sure you want to quit?");

    /* create help widget + callback */

    info = XtVaCreateManagedWidget("Info",
                                     xmCascadeButtonWidgetClass, menu_bar,
                                     XmNmnemonic, 'I',
                                     NULL);

    XtAddCallback(info, XmNactivateCallback, info_pop_up,
                  "Select Prompt Option To Get Program Going.");

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

void prompt_pop_up(Widget cascade_button, char *text,
                  XmPushButtonCallbackStruct *cbs)

```

Table 1. *Continued*

```

void quit_pop_up(Widget cascade_button, char *text,
XmPushButtonCallbackStruct *cbs)

{
    Widget dialog;
    XmString xm_string;
    Arg args[1];

    /* label the dialog */
    xm_string = XmStringCreateLocalized(text);
    XtSetArg(args[0], XmNmessageString, xm_string);
    /* set up default button for cancel callback */
    XtSetArg(args[1], XmNdefaultButtonType,
                XmDIALOG_CANCEL_BUTTON);

    /* Create the WarningDialog */
    dialog = XmCreateWarningDialog(cascade_button, "quit",
                                   args, 1);

    ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

    XmStringFree(xm_string);

    XtAddCallback(dialog, XmNokCallback, quit_activate,
                  NULL);

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);
}

/* routine to remove a DialButton from a Dialog */
void ScrubDial(Widget wid, int dial)
{
    Widget remove;

    remove = XmMessageBoxGetChild(wid, dial);

    XtUnmanageChild(remove);
}

/* callback function for Prompt activate */
void prompt_activate(Widget widget, XtPointer client_data,
XmSelectionBoxCallbackStruct *selection)
{
    Widget dialog;
    Arg args[2];
    XmString xm_string;

    /* compose InformationDialog output string */
    /* selection->value holds XmString entered to prompt */

    xm_string = XmStringCreateLocalized("You typed: ");
    xm_string = XmStringConcat(xm_string, selection->value);

    XtSetArg(args[0], XmNmessageString, xm_string);
    /* set up default button for OK callback */
    XtSetArg(args[1], XmNdefaultButtonType,
                XmDIALOG_OK_BUTTON);

    /* Create the InformationDialog to echo
       string grabbed from prompt */
    dialog = XmCreateInformationDialog(top_wid,
        "prompt_message", args, 2);
}

```

```

    ScrubDial(dialog, XmDIALOG_CANCEL_BUTTON);
    ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);
}

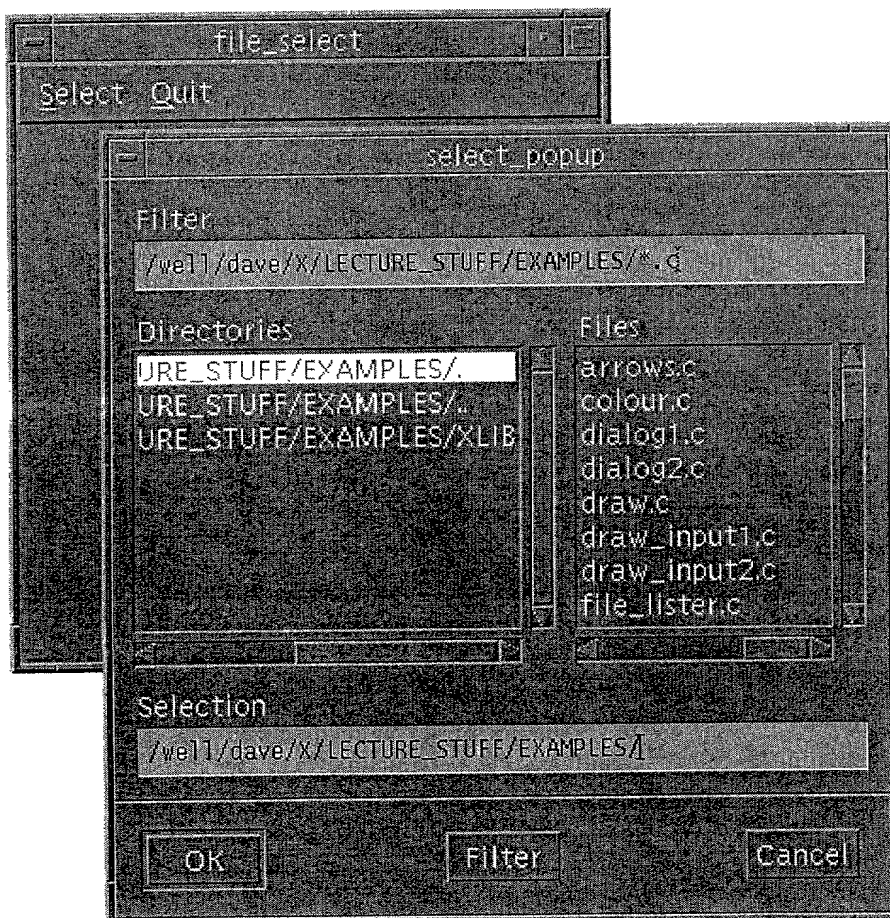
/* callback routines for quit ok dialog */

void quit_activate(Widget dialog) {
    printf("Quit Ok was pressed.\n");
    exit(0);
}

```

## Selection and FileSelection Dialogs

The purpose of both these widgets is to allow the user to select from a list (or in set of lists) displayed within the Dialog. The creation and use of Selection and FileSelection Dialogs is similar. The FileSelectionDialog (Fig. 10.7) allows the selection of files from a directory which has use in many applications (text editors, graphics programs *etc.*) The FileSelection Dialog provides a means for merely browsing directories and selecting file *names*. It is up to the application to read/write the file, or to use the file name appropriately. The SelectionDialog allows for more general selection. We will study the FileSelectionDialog as it is more complex and it is also more commonly used.



## Fig. 10.7 The FileSelectionDialog Widget

To create a FileSelectionDialog, the `XmCreateFileSelectionDialog()` function is commonly used. You must include the `<Xm/FileSB.h>` header file.

A FileSelectionDialog has many resources you can set to control the search of files: (All resources are XmString data types except where indicated.)

### XmNdirectory

-- The directory from where files are (initially) listed. The *default* directory is the *current working directory*.

### XmNdirListLabelString

-- The label displayed above the directory listing in Dialog.

### XmNdirMask

-- The mask used to filter out certain files. *E.g.* in `file_select.c` this mask is set to `*.c` so as only to list C source files in the dialog.

### XmNdirSpec

-- the name of the file (to be) chosen.

### XmNfileListLabelString

-- The label displayed above the file list.

### XmNfileTypeMask

-- The type of file to be listed (*char* type). `XmFILE_REGULAR`, `XmFILE_DIRECTORY`, `XmFILE_TYPE_ANY` are conveniently predefined macros in `<Xm/FileSB.h>`.

### XmNfilterLabelString

-- The label displayed above the filter list.

The search directory, directory mask and others can be altered from within the Dialog window.

The FileSelectionDialog has many child widgets under its control. It is sometimes useful to take control of these child widget in order to have greater control of their resources or callback or even to remove (`XtUnmanageChild()`) one. The function `XmFileSelectionBoxGetChild()` is used to return the ID of a specified child widget. The function takes two arguments:

### Widget

-- the parent widget.

### Child

-- an unsigned char. Possible values for this argument are defined in `<Xm/FileSB.h>` and include:

```

XmDIALOG_APPLY_BUTTON,      XmDIALOG_LIST,
XmDIALOG_CANCEL_BUTTON,     XmDIALOG_LIST_LABEL,
XmDIALOG_DEFAULT_BUTTON,    TXmDIALOG_OK_BUTTON,
XmDIALOG_DIR_LIST,          XmDIALOG_SELECTION_LABEL,
XmDIALOG_DIR_LIST_LABEL,    XmDIALOG_SEPARATOR,
XmDIALOG_FILTER_LABEL,      XmDIALOG_TEXT,
XmDIALOG_FILTER_TEXT,       XmDIALOG_WORK_AREA,
XmDIALOG_HELP_BUTTON.
```

## The file\_select.c program

The program `file_select.c` simply looks for C source files in a directory -- the `XmNdirMask` resource is set to filter out only `*.c` files. If a file is selected it's *listing* is printed to standard output.

```

#include <stdio.h>

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/FileSB.h>

/* prototype callbacks and other functions */
void quit_pop_up(Widget , char *,
                 XmPushButtonCallbackStruct *),
void select_pop_up(Widget , char *,
                  XmPushButtonCallbackStruct *);
void ScrubDial(Widget, int);
void select_activate(Widget , XtPointer ,
                    XmFileSelectionBoxCallbackStruct *)
void quit_activate(Widget)
void cancel(Widget , XtPointer ,
            XmFileSelectionBoxCallbackStruct *);
void error(char *, char *);

File *fopen();

Widget top_wid;

main(int argc, char *argv[])
{
    XtAppContext app;
    Widget main_w, menu_bar, file_select, quit;

    top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
                               &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
                                       xmMainWindowWidgetClass, top_wid,
                                       XmNheight, 300,
                                       XmNwidth, 300,
                                       NULL);

    menu_bar = XmCreateMenuBar(main_w, "main_list",
                               NULL, 0);
    XtManageChild(menu_bar);

    /* create prompt widget + callback */

    file_select = XtVaCreateManagedWidget( "Select",
                                             xmCascadeButtonWidgetClass, menu_bar,
                                             XmNmnemonic, 'S',
                                             NULL);

    /* Callback has data passed to */
    XtAddCallback(file_select, XmNactivateCallback,
                  select_pop_up, NULL);

    /* create quit widget + callback */
    quit = XtVaCreateManagedWidget( "Quit",

```



```

xmCascadeButtonWidgetClass, menu_bar,
XmNmnemonic, 'Q',
NULL);

```

```

/* Callback has data passed to */
XtAddCallback(quit, XmNactivateCallback, quit_pop_up,
               "Are you sure you want to quit?");

```

```

XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

```

```

void select_pop_up(Widget cascade_button, char *text,
                   XmPushButtonCallbackStruct *cbs)

```

```

{
    Widget dialog, remove;
    XmString mask;
    Arg args[1];

    /* Create the FileSelectionDialog */
    mask = XmStringCreateLocalized("*.c");
    XtSetArg(args[0], XmNdirMask, mask);

    dialog = XmCreateFileSelectionDialog(cascade_button,
                                         "select", args, 1);
    XtAddCallback(dialog, XmNokCallback, select_activate,
                  NULL);
    XtAddCallback(dialog, XmNcancelCallback, cancel, NULL);

    remove = XmSelectionBoxGetChild(dialog,
                                     XmDIALOG_HELP_BUTTON);

    XtUnmanageChild(remove); /* delete HELP BUTTON */

    XtManageChild(dialog);
    XtPopUp(XtParent(dialog), XtGrabNone);
}

```

```

void quit_pop_up(Widget cascade_button, char *text,
                  XmPushButtonCallbackStruct *cbs)

```

```

{
    Widget dialog;
    XmString xm_string;
    Arg args[2];

    /* label the dialog */
    xm_string = XmStringCreateLocalized(text);
    XtSetArg(args[0], XmNmessageString, xm_string);
    /* set up default button for cancel callback */
    XtSetArg(args[1], XmNdefaultButtonType,
              XmDIALOG_CANCEL_BUTTON);

    /* Create the WarningDialog */
    dialog = XmCreateWarningDialog(cascade_button, "quit",
                                   args, 2);

    ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

    XmStringFree(xm_string);
}

```

```

        XtAddCallback(dialog, XmNokCallback, quit_activate,
                        NULL);

        XtManageChild(dialog);
        XtPopup(XtParent(dialog), XtGrabNone);
    }

    /* routine to remove a DialButton from a Dialog */

void ScrubDial(Widget wid, int dial)
{
    Widget remove;

    remove = XmMessageBoxGetChild(wid, dial);
    XtUnmanageChild(remove);
}

/* callback function for Prompt activate */

void select_activate(Widget widget, XtPointer client_data,
                    XmFileSelectionBoxCallbackStruct *selection)
{
    /* function opens file (text) and prints to stdout */

    FILE *fp;
    char *filename, line[200];

    XmStringGetLtoR(selection->value,
                    XmSTRING_DEFAULT_CHARSET, &filename);

    if ( (fp = fopen(filename,"r")) == NULL)
        error("CANNOT OPEN FILE", filename);
    else
    {
        while ( !feof(fp) )
        {
            fgets(line,200,fp);
            printf("%s\n",line);
        }
        fclose(fp);
    }
}

void cancel(Widget widget, XtPointer client_data,
            XmFileSelectionBoxCallbackStruct *selection)
{
    XtUnmanageChild(widget); /* undisplay widget */
}

void error(char *s1, char *s2)
{
    /* prints error to stdout */

    printf("%s: %s\n", s1, s2);
    exit(-1);
}

/* callback routines for quit ok dialog */

void quit_activate(Widget dialog)

```

```

{
    printf("Quit Ok was pressed.\n");
    exit(0);
}

```

## User Defined Dialogs

Motif allows the programmer to create new customised Dialogs. `BulletinBoardDialogs` and `FormDialogs` let you place widgets within them in a similar fashion to their corresponding `BulletinBoard` and `Form` widgets. We will, therefore, not deal with these further in this text.

## Exercises

### NEED SOME ON ERROR, WORKING, Question, PROMPT AND OTHER

#### Exercise 8579

Rewrite the `error()` function of the `file_select.c` program so that errors trapped by this program are displayed in an `ErrorDialog` widget and not simply printed to standard output.

## Text Widgets

Text editing is a key task in many applications. For example, Single-line editors are a convenient and flexible means of string data entry for many applications. Indeed, the `FileSelectionDialog` widget (Chapter 10) and other composite widgets have a single text widget as constituent components. More complete multi-line text entry may also be required for many applications.

Motif, conveniently provides a fully functional text widget. This saves the application programmer a lot of work, since tasks such as *cut and paste* editing, text search and insertion are provided within the widget class.

**Note:** Coupled with other advanced facilities such as `FileSelection` widgets *etc.*, we could easily assemble our own fully working text editor application program from component widget classes and little other code.

Motif 1.2 provides two classes of text widgets:

#### Text widget

-- A fully functional multiline windows based text editor.

#### TextField

-- A single line text editor.

Both the above widgets use the (standard C) `string` data type as the base structure for all text operations. This is different from most other Motif widgets. Motif 2.0 provides an additional text widget, `CSText`, which is basically similar to the `Text` widget except that the `XmString` data type is used in text processing.

We will study the `Text` widget in detail in this Chapter. The `TextField` is, essentially, a simpler version of

this and will therefore only be addressed when appropriate. In fact, we can actually make the **Text** widget a single line type by setting the resource `XmNeditMode` to `XmSINGLE_LINE_EDIT`.

## Text Widget Creation

There are a variety of ways to create a Text widget:

- We can use the usual `XmCreateText()` or `XtVaCreateManagedWidget()` methods.
- Usually we will need to use a *scrolled* window since the text we are editing may exceed the window size (Fig 11.1).
  - To create a `ScrolledText` widget use `XmCreateScrolledText()`.
- We must include the `<Xm/Text.h>` header file for all Text widget applications. There are corresponding `<Xm/TextF.h>` and `<Xm/CSText.h>` header files for the `TextField` and `CSText` widgets respectively.

There are various resources that can be usefully set for a Text widget:

### **XmNrows**

-- The visible number of rows.

### **XmNcolumns**

-- The visible number of columns.

### **XmNeditable**

-- True or False: determines whether editing of displayed text is allowed.

### **XmNscrollHorizontal**

-- True or False: Turn scrolling On/Off in this direction.

### **XmNscrollVertical**

-- True or False: Turn scrolling On/Off in this direction.

## Putting text into a Text Widget

The Text widget is a *dynamic* structure and text may be inserted into the widget at any time. There are many text editing and insertion functions that will be introduced shortly. The simplest operation is actually setting the text that will be used by the Text widget.

The function `XmTextSetString()` puts a specified *ordinary (C type)* string into a specified widget. It has two arguments:

### **Widget**

-- the Text widget,

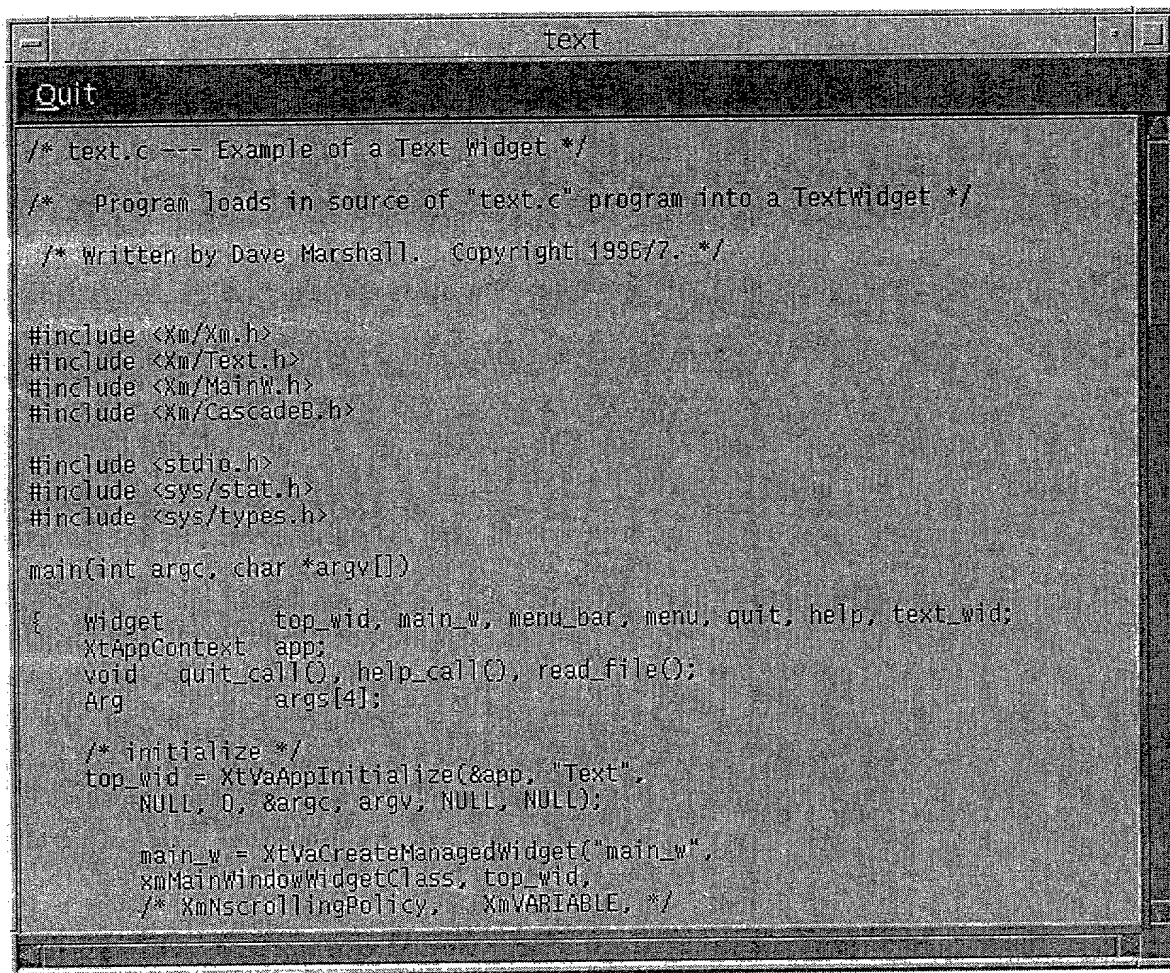
### **String**

-- The text being placed in the widget.

## Example Text Program - `text.c`

This program is a fairly simple example of the Text widget in use.

- It creates a `ScrolledText` widget and places the source code of the `text.c` program in the Text widget (Fig. 11.1).
- **No** editing is allowed.



**Fig. 11.1** ScrolledText Widget

```

#include <Xm/Xm.h>
#include <Xm/Text.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

/* Prototype Callback and other functions */

void quit_call(),
     help_call(),
     read_file(Widget);

main(int argc, char *argv[])
{
    Widget      top_wid, main_w, menu_bar,

```

[illegible]

{

```
return;
```

```

    if (!(text = XtMalloc((unsigned)(statb.st_size+1))))
    {
        fprintf(stderr, "Can't alloc enough space for %s",
                filename);
        XtFree(filename);
        fclose(fp);
        return;
    }

    if (!fread(text, sizeof(char), statb.st_size+1, fp))
        fprintf(stderr, "File read error\n");

    text[statb.st_size] = 0; /* be sure to NULL-terminate */

    /* insert file contents in TextWidget */

    XmTextSetString(text_wid, text);

    /* free memory
    */
    XtFree(text);
    XtFree(filename);
    fclose(fp);
}

void quit_call()
{
    printf("Quitting program\n");
    exit(0);
}

```

## Editing Text

Motif provides many functions that allow the editing of the text (string) stored in the widget. Text can be searched, inserted and replaced.

To replace all or parts of the text in a Text widget use the `XmTextReplace()` function. It has four arguments:

### Widget

-- The Text widget.

### Start Position (`XmTextPosition` type)

-- A long int measured in bytes. Specifies where to begin inserting text.

### End Position (`XmTextPosition` type)

-- The end insertion point.

### New text

-- The string that will replace existing text.

No matter how long the specified replacement text string is, text is *only* replaced (character-by-character) between the 2 positions. However, if the start and end positions are equal then text is inserted after the given position.

An alternative method to insert text, is to use the `XmTextInsert()` function. This takes 3 arguments:

### Widget

-- the Text Widget,

**Insert Position**

-- the XmTextPosition for the insertion,

**Insertion Text**

-- the text String.

To Search for a string in the Text widget , use the XmTextFindString() function with the following arguments:

**Text Widget**

-- to be searched,

**Start Position**

-- as before,

**Search string,****Search Direction**

-- either XmTEXT\_FORWARD or XmTEXT\_BACKWARD,

**Position**

-- a XmTextPosition pointer that returns the position found.

XmTextFindString() returns a Boolean value which is False if no string was found.

To obtain text (in full) from a Text widget use the XmTextGetString() function to return a String for a specified widget. An example use of this function is to save text stored in the Text widget to a file. This can be simply achieved by:

- XmTextGetString() from the widget.
- Write the string (e.g. fprintf()) to a file.

The function XmTextGetSubstring() can be used to get a portion of text from a widget. It takes 5 arguments:

**Text Widget**

-- from where the substring is to be obtained.

**Start Position**

-- the XmTextPosition of the first character in the substring to be returned.

**Number of Characters**

-- to be copied from the substring.

**Buffer Size**

-- the number of characters in the String buffer where the substring is copied to.

**String**

-- a pointer to a String buffer which will hold the substring value when this function is returned.

Similar functions exist for both the TextField and the CStext widgets. An example of these functions in use with the TextField widget is given in Section [11.7](#).

## Scrolling Control

Motif provides a variety of functions to control the scrolling of the text within a Text widget. Thus, the application programmer has control over which portions of text can be displayed at a given time. The



following options are available:

- To show a piece of text in a given position, use `XmTextShowPosition()`.
- To set a line to be at the top of scrolled widget, specify the position to `XmTextSetTopCharacter()`.
- To position the cursor to a position where text may be inserted, use `XmTextSetInsertionPosition()`.

Similar functions exist for both the `TextField` and the `CSText` widgets. An example of these functions in use with the `TextField` widget is given in Section [11.7](#).

## Text Callbacks

Behind almost all of the Text widget functions, described above, lie default callback resources. These control the basic text editing facilities: cut and paste, searching *etc.* However, there may be occasions when the application may need greater control over things. Several callback resources are provided for this purpose. Briefly these are:

### **XmNactivateCallback**

-- Called on **Enter** key press (only single-line Text widgets).

### **XmNverifyCallback**

-- Used to verify a change to text *before* it is made.

### **XmNvalueChangedCallback**

-- Called *after* a change to text has been made.

### **XmNmotionCallback**

-- Called when the user has altered the cursor position, or made a text selection with mouse.

### **XmNfocusCallback**

-- Called when the user wants to begin input.

### **XmNlosingfocusCallback**

-- Called when the widget is losing keyboard focus.

We can use *verifyCallbacks* for checking user inputs -- for example for password verification.

## Editing and Scrolling in Practice

The `test_for_echo.c` program (Section [9.3](#)) illustrates the use of some of the editing and scrolling functions described. The program basically operates as follows:

- Two `TextField` widgets are created that are then made the command and message areas of a `MainWindow` widget (Fig. [9.6](#)).
- The command area `Textfield` is allowed to receive input but the message area is not. The `XmNeditable` resource is set to `False` for the message area widget, `msg_wid` in `test_for_echo.c`.
- After the widgets are instantiated the command and message area prompts are put into the respective widgets using the `XmTextFieldSetString()` function. The text insertion point is set for the command area so that input can be received after the command area prompt.
- When the command area receives a `XmNactivateCallback` event, a callback `cmd_cbk`, is called that

parses the command area for a substring (excluding the command area prompt) using the `XmTextFieldGetSubstring()` function.

- The substring is then appended to the message area after the message area prompt using the `XmTextFieldReplace()`, with the start point set appropriately.
- The command area prompt is then reset by simply setting (using the `XmTextFieldSetString()` function) it to the original prompt value.
- The insertion point for the command area is also reset (using the `XmTextFieldSetInsertionPosition()` function) for subsequent data input.

## Exercises

### Exercise 8623

Modify the `text.c` (Section 11.3) so that it employs a `FileSelection` widget to allow the user to select a file, which it then reads and displays in a `ScrolledText` Widget.

### MORE EXERCISES

## List Widgets

The `List` widget allows selection from a variety of specified items. The `List` widget is actually one of the component widgets in the `FileSelectionBox` widget (Chapter 10).

The use of a `List` is similar to that of a `Menu`, but is a little more flexible:

- We can add and delete items from the list.
- Lists can be scrolled.
- A number of Selection modes are available.

An example of a `List` Widget is shown in Fig. 12.1.



Fig. 12.1 Output of `list.c`

## List basics

To create a simple list use: `XtVaCreateManagedWidget()`, and specify `xmListWidgetClass` as the widget type (or use `XmCreateList()`). The header file `<Xm/List.h>` must be included. We will usually want to create a `ScrolledList`. To do this use: `XmCreateScrolledList()`.

There are a number of useful resources:

### **XmNitemCount**

-- The number of items in the list.

### **XmNitems**

-- The item list. The item list is a `XmStringTable` data type. This is basically a 1D array of `XmStrings`.

### **XmNselectionPolicy**

-- Controls how items are chosen (*see* Section [12.1.1](#) below).

### **XmNvisibleItemCount**

-- The number of items shown in the list. This determines height of widget.

### **XmNscrollBarDisplayPolicy**

-- Either `XmAS_NEEDED` or `XmSTATIC`. `XmSTATIC` will always show (vertical) scroll even if all items are visible.

### **XmNlistSizePolicy**

-- `XmCONSTANT`, `XmRESIZE_IF_POSSIBLE` or `XmVARIABLE`. Controls horizontal scrolling.

### **XmNselectedItemCount**

-- The number of selected items.

### **XmNselectedItems**

-- The selected items from the list. These can be set at list initialisation to enable a *default* choice to be specified.

## **List selection modes**

One primary distinction between a list and menu is in the methods of selection available. Four types of selection are available. The `List` resource `XmNselectionPolicy` must be set accordingly:

### **Single**

-- `XmSINGLE_SELECT` (defined in `<Xm/List.h>`). Only one item may be selected.

### **Browse**

-- `XmBROWSE_SELECT`. Similar to `Single` selection, except a default selection can be provided and user interaction may vary.

### **Multiple**

-- `XmMULTIPLE_SELECT`. More than one item may be selected. A mouse click on a item will select it. If you click on an already selected item it will be *deselected*.

### **Extended**

-- `XmEXTENDED_SELECT`. Like `Multiple` selection. Here you can drag the mouse over a number of items to select them.

## **Adding and removing list items**

As the name of this widget implies, the *List* is a dynamic structure that can grow or shrink as items are added or deleted.

To add an item to a list, use the function `XmListAddItem()`, which takes 3 arguments:

## The List Widget

-- The widget we add items to,

### Item

-- An (XmString) list item we wish to add,

### Position

-- The (int) position. **Note:** List indexing **Starts at index 1** (Die hard C programmers take note). Position **0** in the list is used to specify the last position in the list. If you specify a **0** position items will get appended to the end of the List.

Another function `XmListAddItemUnselected()` has exactly the same syntax as `XmListAddItem()`, above. This function will guarantee that an item is not selected when it is added. This is not always the case with the `XmListAddItem()`, since selection will be dependent on the currently selected list index.

To remove a *single* named (XmString) item, `str`, from a List widget, use the `XmListDeleteItem(Widget List, XmString str)` function.

To remove a number of named (XmString) items, use the `XmListDeleteItems(Widget List, XmString *del_items)` function where the second argument, `del_items`, is an array of XmStrings that contain the names of items being deleted.

If you know the position of item(s) in a List, as opposed to their names, you can use the following functions:

```
XmListDeletePos(Widget wid, int pos)
    where the second pos argument specifies the deletion position.
XmListDeleteItemsPos(Widget wid, int num, int pos)
    where num items are deleted starting from position pos.
```

To delete *all* items from a list, use `XmListDeleteAllItems(Widget wid)`.

## Selecting and Deselecting items

Two functions `XmListSelectItem(Widget, XmString, Boolean)` and `XmListSelectPos(Widget, int, Boolean)` may be used to select an item from within a program.

The Boolean value, if set to `True`, will call the callback function associated with the particular List.

To deselect items use `XmListDeselectItem(Widget, XmString)`, `XmListDeselectPos(Widget, int)` or `XmListDeselectAllItems(Widget)`. The operation of which is similar to corresponding delete functions.

## List Enquiry

Since the List is *dynamic* we may need to know how long the list is, and which items are currently selected *etc.*. Some of these values can be obtained from the callback structure of a list (*see* Section [12.3](#) below). However, if no callback has been invoked the programmer may sometimes still need to access this

information.

The List resources are updated automatically (by default callback resources) so all we need to do is to `XtGetValues()` (or something similar) for the resource we want. For example:

- To find the length of a list:

Obtain the value of the resource `XmNItemCount`.

```
Arg args[1]; /* Arg array */
int n = 1; /* number arguments */
int list_size; /* value to store XtGetValue() request */

.....

XtSetArg(args[0], XmNItemCount, &list_size);
XtGetValues(list_wid, args, n);

printf("The Size of the list = %d\n", list_size);
```

where `list_wid` is the List widget we request this information from.

**Note:** We pass the address of `list_size` to `XtSetArg()` since we need to specify a pointer to physical (program) memory in which to store the result. The value of `list_size` is available after the `XtGetValues()` call and is only *accurate* until the next user (or application) list addition/subtractions.

- To find the number of items currently selected:


Obtain the value of the resource `XmNselectedItemCount`:

```
Arg args[1]; /* Arg array */
int n = 1; /* number arguments */
int select_count; /* value to store
                  XtGetValue() request */

.....

XtSetArg(args[0], XmNselectedItemCount, &select_count);
XtGetValues(list_wid, args, n);

printf("The Numver of selected list items = %d\n",
select_count);
```

Recall that the use of `XtGetValue()` is similar to that of `XtSetValue()` (Chapter )

## List Callbacks

Default List callback functions facilitate common interaction with a List such as selection of an item (or multiple items) and addition or deletion of items. More importantly, related resource information is automatically updated (e.g. the current List size, `XmNItemCount`). There is a List callback resource for each of the selection types (e.g. `XmNsinglSelectionCallback`) and also a `XmNdefaultActionCallback`. The application programmer is free to add his own callback functions in the usual manner. In this case, the selection policy callback will be called first and then the default.

The Callback function has the usual form:

```
list_cbk(Widget w, XtPointer data, XmListCallbackStruct *cbk)
```


Elements of the `XmListCallbackStruct` include:

```
item
    -- the XmString of the selection.
item_position
    -- position in the List.
selected_items
    -- the List of XmStrings in multiple selections.
selected_item_count
    -- number of multiple selections.
selected_item_positions
    -- positions in the List.
```

An example of the use of a List callback is given in the following `list.c` example program.

## The `list.c` program

An example of a List in action is given in `list.c`.

We create a simple list that shows a selection of colours. Selection of these colours changes the background colour  of the List widget.

```
#include <Xm/Xm.h>
#include <Xm/List.h>

/* Prototype Callback */

void list_cbk(Widget , XtPointer ,
              XmListCallbackStruct *);

String colours[] = { "Black", "Red", "Green",
                    "Blue", "Grey"};

Display *display; /* xlib id of display */
Colormap cmap;

main(int argc, char *argv[])
{
    Widget          top_wid, list;
    XtAppContext    app;
    int             i, n = XtNumber(colours);
    XColor          back, fore, spare;
    XmStringTable    str_list;
    Arg             args[4];
```

```

top_wid = XtVaAppInitialize(&app, "List_top", NULL, 0,
    &argc, argv, NULL, NULL);

str_list =
    (XmStringTable) XtMalloc(n * sizeof (XmString *));

for (i = 0; i < n; i++)
    str_list[i] = XmStringCreateSimple(colours[i]);

list = XtVaCreateManagedWidget("List",
    xmListWidgetClass,      top_wid,
    XmNvisibleItemCount,   n,
    XmNitemCount,          n,
    XmNitems,              str_list,
    XmNwidth,              300,
    XmNheight,             300,
    NULL);

for (i = 0; i < n; i++)
    XmStringFree(str_list[i]);
XtFree(str_list);

/* background pixel to black foreground to white */

cmap = DefaultColormapOfScreen(XtScreen(list));
display = XtDisplay(list);

XAllocNamedColor(display, cmap, colours[0], &back,
    &spare);
XAllocNamedColor(display, cmap, "white", &fore, &spare);

n = 0;
XtSetArg(args[n], XmNbackground, back.pixel);
++n;
XtSetArg(args[n], XmNforeground, fore.pixel);
++n;
XtSetValues(list, args, n);

XtAddCallback(list, XmNdefaultActionCallback, list_cbk,
    NULL);

XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

/* called from any of the "Colour" list items.
Change the color of the list widget.
Note: we have to use dynamic setting with XtSetValues()..
*/
void list_cbk(Widget w, XtPointer data,
    XmListCallbackStruct *list_cbs)
{
    int n = 0;
    Arg args[1];
    String selection;
    XColor xcolour, spare; /* xlib color struct */

    /* list->cbs holds XmString of selected list item */
    /* map this to "ordinary" string */

    XmStringGetLtoR(list_cbs->item, XmSTRING_DEFAULT_CHARSET,
        &selection);

```

```

    if (XAllocNamedColor(display, cmap, selection,
                        &xcolour, &spare) == 0)
        return;

    XtSetArg(args[n], XmNbackground, xcolour.pixel);
    ++n;
    /* w id of list widget passed in */
    XtSetValues(w, args, n);
}

```

## Exercises

Needs SOME

## The Scale Widget

The Scale widget allows the user to input numeric values into a program. An example of the Scale widget is shown in Fig. 13.1.

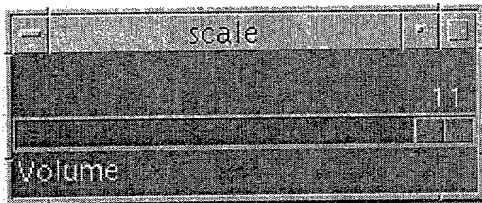


Fig. 13.1 Output of scale.c

## Scale basics

To create a Scale Widget use `XtVaCreateManagedWidget()`, with a `xmScaleWidgetClass` class pointer or use `XmCreateScale()`. Once again a header file, `<Xm/Scale.h>`, needs to be included in all programs using Scale widgets.

The following Scale Widget Resources are typically used:

### **XmNmaximum**

-- The largest value of scale.

### **XmNminimum**

-- The scale's smallest value. The default value is 0.

### **XmNorientation**

-- `XmHORIZONTAL` or `XmVERTICAL`, resource values define how the Scale is displayed.

### **XmNtitleString**

-- The `xmString` label of the scale.

### **XmNdecimalPoints**

-- A scale value is always returned as an integer (default 0). This resource can be set to give the



user the impression of floating point input, *e.g.* if we have a range 0 - 1000 on the Scale but `XmNdecimalPoints` set to 2. The displayed range would be 0.00 - 10.00.

The application programmer must take care of the input value to the program. In the above a value will still get returned in the integer range and somewhere in the application there must be a division by 100.

### **XmNshowValue**

-- True or False. This resource decides whether or not to display the value of the scale as it moves.

### **XmNvalue**

-- The current value (int) of the Scale.

### **XmNprocessingDirection**

-- Either `XmMAX_ON_TOP`, `XmMAX_ON_BOTTOM`, `XmMAX_ON_LEFT` or `XmMAX_ON_RIGHT`. This resource sets the end of the scale, where the maximum and minimum values are placed. This depends on the orientation of the Scale.

## Scale Callbacks

The Scale callback can be called for two types of events:

### **XmNvalueChangedCallback**

-- If the value is changed.

### **XmNdragCallback**

-- If the Scale value is moved at all, "continuous" values can be input. **Note:** This may affect X performance as it involves *instantaneously* updating the display of the scale.

The Scale callback function is standard:

```
void scale_cbk(Widget w, XtPointer data,
               XmScaleCallbackStruct *struct)
```

The structure element `value` holds the current Scale **integer** value, and is the only structure element that is really of interest. The `scale.c` program illustrates this usage.

## The `scale.c` program

The program simply brings up a virtual *volume* Scale (in the context of a virtual amplifier controller). The user changes the value which is caught by a `XmNvalueChangedCallback` and the current value is

interrogated to print a message to standard output.

```
#include <Xm/Xm.h>
#include <Xm/Scale.h>

/* Prototype callback */

void scale_cbk(Widget , int ,
               XmScaleCallbackStruct *);

main(int argc, char **argv)
{
    Widget      top_wid, scale;
    XmString     title;
    XtAppContext app;

    top_wid = XtVaAppInitialize(&app, "Scale_eg", NULL, 0,
                               &argc, argv, NULL, NULL);

    title = XmStringCreateLocalized("Volume");

    scale = XtVaCreateManagedWidget("scale",
                                     xmScaleWidgetClass, top_wid,
                                     XmNtitleString, title,
                                     XmNOrientation, XmHORIZONTAL,
                                     XmNmaximum, 11,
                                     XmNdecimalPoints, 0,
                                     XmNshowValue, True,
                                     XmNwidth, 200,
                                     XmNheight, 100,
                                     NULL);

    XtAddCallback(scale, XmNvalueChangedCallback, scale_cbk,
                  NULL);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

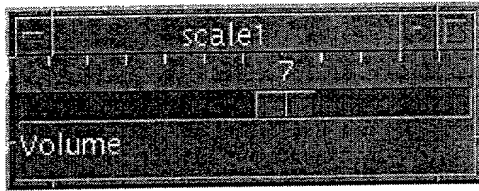
void scale_cbk(Widget widget, int data,
               XmScaleCallbackStruct *scale_struct)
{
    if (scale_struct->value < 4)
        printf("Volume too quiet (%d)\n");
    else if (scale_struct->value < 7)
        printf("Volume Ok (%d)\n");

    else
        if (scale_struct->value < 10)
            printf("Volume too loud (%d)\n");
        else /* Volume == 11 */
            printf("Volume VERY Loud (%d)\n");
}
```

## Scale Style

### Motif 1.2 Style

The *Motif Style Guide* (Chapter 20) suggests that some sort of markers should be used to gauge distance along a Scale. However, no provision is made for this within the Scale widget class in Motif 1.2. Instead, the programmer must assemble this manually. An assortment of labels and tics can be used to provide some sort of visual *ruler* (Fig. 13.2).



**Fig. 13.2 Better Style Scale Output**

The Motif program code that achieves this, by placing vertical SeparatorGadgets ("|") at equally spaced intervals, is as follows:

```
Widget ...,tics[1];
.....
.....
/* label scale axis */
for (i=0; i < 11; ++i )
{
    XtSetArg(args[0], XmNseparatorType, XmSINGLE_LINE);
    XtSetArg(args[1], XmNOrientation, XmVERTICAL);
    XtSetArg(args[2], XmNwidth, 10);
    XtSetArg(args[3], XmNheight, 5);
    tics[i] = XmCreateSeparatorGadget(scale, "|",
                                     args, 4);
}

XtManageChildren(tics, 11);
.....
.....
```

## The Motif 2.0 Style

Motif 2.0 provides a new convenience function `XmScaleSetTicks()` to allow for an easier configuration of ticks along the Scale widget. The configuration allows for three different sized ticks to be placed at specified regular intervals along the Scale. Each tick mark is actually a SeparatorGadget oriented perpendicular to the Scale's orientation. The function `XmScaleSetTicks()` takes seven arguments:

### Widget

-- The scale widget being assigned the ticks.

### int large\_every

-- The number of Scale values between large ticks.

### int num\_medium

-- The number of medium ticks between large ticks.

### int num\_small

-- The number of small ticks between medium ticks.

### Dimension size\_large

-- The size (width or height) of the big ticks.

**Dimension** `size_medium`

-- The size (width or height) of the medium ticks.

**Dimension** `size_small`

-- The size (width or height) of the small ticks.

If you specify tick marks for a Scale and then change the Scale's orientation then you must remove all the tick marks and then recreate new ones in the correct orientation. This may be achieved by the following method:

- To remove ticks marks you must destroy all SeparatorGadget tick mark children:
  - The first two children of a Scale are its title and scroll bar.
  - *All* additional children are the tick marks.
  - The function `XtDestroyChildren()` can be used to remove specified tick marks.
- Call `XmScaleSetTicks()` with appropriate arguments.

## Exercises

NEEDS SOME

## ScrolledWindow and ScrollBar Widgets

We have already seen scrollbars in action with Text (Chapter 11) and List widgets (Chapter 12). More generally, Motif provides a ScrolledWindow widget that allows scrolling of any widget contained within it. This means that we can place a large view area inside a smaller one and then view portions of the view area.

As we have seen, Motif actually provides convenience functions to produce ready made ScrolledText and ScrolledList widgets. These are, in fact, Text or List widgets contained inside a ScrolledWindow widget.

ScrollBars are the basic components of scrolling. A ScrolledWindow widget may contain either, or both of, horizontal and vertical ScrollBars. Many application will typically use ScrollBars. In the majority of instances, ScrollBar widgets will be created automatically by higher level widgets (e.g. ScrolledWindow, ScrolledText or ScrolledList). Occasionally, greater control over the default settings of the ScrollBar will be required. Sometimes, these resources can be set as resources of the higher level widget, other times the resources will need to be set explicitly. In this chapter, we will highlight important ScrollBar resources and illustrate how they can be set in both of the above scenarios.

## ScrolledWindow Widgets

ScrolledWindow widgets can be created manually with `XtVaCreateManagedWidget()`, with the `xmScrolledWindowWidgetClass` pointer or with `XmCreateScrolledWindow()` function.

Associated definitions for this widget class *etc.* are included in the `<Xm/ScrolledW.h>` header file.

Useful resources include:

### **XmNhorizontal, XmNvertical**

-- The identifiers (IDs) of component ScrollBar widgets of the ScrolledWindow. One or both may be present.

### **XmNscrollingPolicy**

-- Either defined as `XmAUTOMATIC` or `XmAPPLICATION_DEFINED`. If the scrolling policy is `XmAUTOMATIC` then scrolling is taken care of otherwise the application must create and control `XmScrollBar` widgets itself.

### **XmNscrolBarDisplayPolicy**

-- Either defined as `XmAS_NEEDED` or `XmSTATIC`, as with List Scrolling.

### **XmNvisualPolicy**

-- Either defined as `XmCONSTANT` or `XmVARIABLE`. If constant then scrolled window cannot resize itself.

### **XmNworkWindow**

-- The widget to be scrolled.

## ScrollBar Widgets

You may have to create ScrollBars yourself, especially if the scrolling policy is defined as `XmAPPLICATION_DEFINED`. Alternatively, you may get the ID of a ScrollBar from a ScrolledWindow (e.g. `XmNhorizontal` resource).

To create a ScrollBar, use `XtVaCreateManagedWidget()` with `xmScrollBarWidgetClass` or use `XmCreateScrollBar()`.

To obtain a horizontal ScrollBar ID from a ScrolledWindow:

```
Arg args[1]; /* Arg array */
int n = 1; /* number arguments */
Widget scrollwin, scrollbar;

scrollwin = XmCreateScrolledWindow(.....)
.....

XtSetArg(args[0], XmNhorizontal, &scrollbar);
XtGetValues(scrollwin, args, n);
```

Typical ScrollBar resources include:

### **XmNsliderSize**

-- A slider may be divided up into *unit lengths*. This resource sets its size.

### **XmNmaximum**

-- The largest size (measured in unit lengths) a ScrollBar can have.

### **XmNminimum**

-- The smallest ScrollBar size.

### **XmNincrement**

-- The number of unit lengths the Scale will change when moved with mouse.

## XmNorientation

-- XmVERTICAL (Default) or XmHORIZONTAL layout of the widget.

## XmNvalue

-- The current position of the Scale.

## XmNpageIncrement

-- Controls how much the underlying *work window* moves relative to a ScrollBar movement.

The Callback resources for a ScrollBar are:

## XmNvalueChangedCallback

-- Called if the ScrollBar value changes.

## XmNdecrementCallback, XmNincrementCallback

-- Called if the ScrollBar value changes up or down.

## XmNdragCallback

-- Called on *continuous* Scale value updates.

## XmNpageDecrementCallback, XmNpageIncrementCallback

-- Called on movement of *work window*.

## XmNtoTopCallback, XmNtoBottomCallback

-- Called if ScrollBar is moved to minimum or maximum values.

# Exercises

NEEDS SOME????????

# Toggle Widgets

A Toggle Widget basically provides a simple switch type of selection. The Toggle is either a square or circle shape indicator which if pressed can be turned on and if pressed again turned off. Text and pictorial items (*pixmap*s) can be used to label a Toggle .

Several Toggle widgets can be grouped together to allow greater control of selection. Motif provides *two* methods of grouping Toggles together.

## RadioBox Widget

-- Only one of the group can be *on* at any given time. Toggles are diamond (Motif 1.2) or circular



(CDE 1.0 and Motif 2.0) in this widget (Fig 15.1).

**Fig. 15.1 A RadioBox set of Toggle Widgets****CheckBox Widget**

-- More than one Toggle can be *on* at any time. Toggles are square. (Fig. 15.2).

**Fig. 15.2 A CheckBox set of Toggle Widgets**

## Toggle Basics

To create a single Toggle use `XtVaCreateManagedWidget()` with a `xmToggleButtonWidgetClass` pointer or use `XmCreateToggleButton()`.

The header file `<Xm/ToggleB.h>` holds definitions *etc* for this widget.

Several Toggle Resources are relevant to the programmer:

**XmNindicatorType**

-- Defines the mode of operation of the Toggle. Set this resource to `XmN_OF_MANY` for a CheckBox or `XmONE_OF_MANY` for a RadioBox.

**XmNindicatorOn**

-- Set this resource to True to turn indicator on. The *default* is False (off).

**XmNindicatorSize**

-- Changes indicator size. Size is specified in Pixel unit size.

**XmNlabelType**

-- Either set to `XmSTRING` or `XmPIXMAP`. Defines the type of label associated with the Toggle.

**XmNlabelString**

-- The `xmString` label of Toggle.

**XmNlabelPixmap**

-- The Pixmap of an unselected Toggle.

**XmNselectPixmap**

-- The Pixmap of a selected Toggle.

**XmNselectColour**

-- The colour of a selected Toggle. (Pixel data type).

The Pixmap is a standard X data type. You can use `xmGetPixmap()` to load in a Pixmap from a file. (See Chapter 16 on Graphics and Xlib for further details.)

## Toggle Callbacks

Toggle Callbacks have the usual callback function format and are prototyped by:

```
void toggle_cbk(Widget, XtPointer,
                XmToggleCallbackStruct *).
```

Toggle Callbacks are activated upon an `XmNvalueChangedCallback`. The `XmToggleCallbackStruct` has a boolean element `set` that is `True` if the Toggle is **on**. The `toggle.c` (Section [15.3](#) below) illustrates the use of Toggle callbacks.

## The toggle.c program

This program creates two `CheckBox` Toggles in a `RowColumn` Widget. When their Callbacks are activated, the state of the Toggle is interrogated and a message is printed to standard output.

```
#include <Xm/Xm.h>
#include <Xm/ToggleB.h>
#include <Xm/RowColumn.h>

/* Prototype callback */
void toggle1_cbk(Widget, XtPointer,
                 XmToggleButtonCallbackStruct *),
void toggle2_cbk(Widget, XtPointer,
                 XmToggleButtonCallbackStruct *);

main(int argc, char **argv)
{
    Widget toplevel, rowcol, toggle1, toggle2;
    XtAppContext app;

    toplevel = XtVaAppInitialize(&app, "Toggle", NULL, 0,
                                &argc, argv, NULL, NULL);

    rowcol = XtVaCreateWidget("rowcol",
                              xmRowColumnWidgetClass, toplevel,
                              XmNwidth, 300,
                              XmNheight, 200,
                              NULL);

    toggle1 = XtVaCreateManagedWidget("Dolby ON/OFF",
                                       xmToggleButtonWidgetClass, rowcol, NULL);

    XtAddCallback(toggle1, XmNvalueChangedCallback,
                  toggle1_cbk, NULL);

    toggle2 = XtVaCreateManagedWidget("Dolby B/C",
                                       xmToggleButtonWidgetClass, rowcol, NULL);

    XtAddCallback(toggle2, XmNvalueChangedCallback,
                  toggle2_cbk, NULL);
    XtManageChild(rowcol);
    XtRealizeWidget(toplevel);
    XtAppMainLoop(app);
}

void toggle1_cbk(Widget widget, XtPointer client_data,
                 XmToggleButtonCallbackStruct *state)
{
    printf("%s: %s\n", XtName(widget),
          state->set? "on" : "off");
}
```



```

}

void
toggle2_cbk(Widget widget, XtPointer client_data,
XmToggleButtonCallbackStruct *state)

{ printf("%s: %s\n", XtName(widget), state->set ? "B" : "C");
}

```

## Grouping Toggles

You can, if you wish, group RadioBoxes or CheckBox Toggles in RowColumn, or Forms yourself (as has been done in the above `toggle.c` program).

However, Motif provides a few convenience functions, `XmCreateSimpleRadioBox()` and `XmCreateSimpleCheckBox()` are common examples.

Basically, these are RowColumn Widgets with Toggle children created automatically. Appropriate resources can be set, e.g. `XmNindicatorType` or `XmNradioBehaviour` (in this enhanced RowColumn Widget).

## Exercises

NEED SOME

## Xlib and Motif

This Chapter will deal specifically with the introduction of **Xlib** - *the low level X library*. Recall that Xlib provides the means of communication between the application and the X system. The Xlib library of (C) subroutines is large and of a comparable size to Motif. Many of Xlib's routines deal with the creation, maintenance and interaction between windows and applications. Xlib does not have any concept of widgets and thus does not provide any (high level) means of interaction. In general, writing complete application solely in Xlib is not a good idea. Motif provides many useful, complete GUI building blocks that should always be used if available. For example, do you really want to write a complete text editing library in Xlib, when Motif provides one for free?

If you use Motif then there should never be any need to resort to Xlib for window creation ☒. Motif is far more powerful and flexible. Consequently, in this and forthcoming Chapters, we will only deal with issues that affect the interfacing of Xlib with Motif and the Xt toolkit.

However, for certain tasks we will have to resort to Xlib. The sort of tasks that we will be concerned with in this text are:

### Graphics

-- The responsibility of actually drawing items to a window is the domain of Xlib. Whilst Motif

does provide a *canvas* where graphics can be drawn, Motif has to rely on Xlib functions to actually draw something. Xlib only facilitates simple 2D graphics.

### Pixmap

-- A Pixmap is an off-screen drawable area where graphics may be placed. Pixmap are clearly related to usage with Xlib graphics. However, Pixmap are also used with Images and also to *label* graphics based widgets *e.g. DrawnButton and Toggle Widgets*.

### Colour

-- Handling colour is a fundamental element for a windowing system. Colour plays an important part in any GUI for providing a user friendly GUI layout, indicating key features and alerting the user to certain actions. However, in the X system colour can be quite complex as a variety of devices are required to be supported by the X network and each device may have a completely different interpretation of colour. Because of this requirement, colour needs to be handled at the Xlib level. Chapter 18 deals with the major issues of colour and X.

### Text

-- Just like colour handling of Text can vary across devices. The font, typeface (*e.g. italic, bold*) and size may need to be controlled by Xlib.

### Events

-- Whilst Motif handles events in a robust and efficient manner, the Motif method of event handling is sometimes a little restrictive. In this case responsibility for event handling is sometimes handed over to Xlib.

## Xlib Basics

In Chapter 3 we described the X Window system and explained the relevance of each system component. We briefly mentioned that Xlib provides the interface between the X Protocol and the application program. Xlib therefore has to deal with many low level tasks. In short, Xlib concerns itself with:

- Display and Server Communication,
- Event and Error handling,
- Window Management -- communication with the window manager,
- Text -- fonts, size style *etc.*,
- Graphics -- 2D graphics routines,
- Colour.

Xlib deals with much lower level objects than widgets. When you write or draw in Xlib reference, may be made to the following:

### Display

-- This structure is used by nearly all Xlib functions. It contains details about the server and its screens.

### Drawable

-- this is an identifier to a structure that can contain graphics (*i.e.* it can be **drawn** to). There are two common types:

#### Window

-- A part of the screen.

#### Pixmap

-- An *off-screen* store of graphical data.

### Screen

-- Information about a *single* screen of a display.

### Graphics Context (GC)

-- When we draw we need to specify things like line width, line style (*e.g* solid or dashed), colour, shading or filling patterns *etc.*. In Xlib we pass such information to a drawing routine via a GC structure. This must be created (and values set) before the routine uses the GC. More than one GC can be used.

### Depth

-- the number of bits per pixel used to display data.

### Visual

-- This structure determines how a display handles screen output such as colour and depends on the number of bits per pixel.

If we are programming in Xlib alone, we would have to create windows and open displays ourselves (*see* [Mar96]) for details). However, if we are using a higher level toolkit such a Motif and require to call on Xlib routines then we need to obtain the above information from an appropriate widget in order pass on appropriate parameter values in the Xlib function calls.

Functions are available to obtain this information readily from a Widget. For example `XtDisplay()`, `XtWindow()`, `XtScreen()` *etc.* can be used to obtain the ID of a given Xlib Display, Window, or Screen structure respectively from a given widget. *Default* Values of these structures are also typically used. Functions `DefaultDepthofScreen()`, `RootWindowofScreen()` are examples.

Sometimes, in a Motif program, you may have to create an Xlib structure from scratch. The GC is the most frequently created structure that concerns us. The Function `XCreateGC()` creates a new GC data structure.

We will look at the mechanics of assembling Xlib graphics within a Motif program when we study DrawingAreas in Chapter 17. For the remainder of this Chapter we will continue to introduce basic Xlib concepts. In the coming Sections, reference is made to programs that are described in Chapter 17.

## Graphics Contexts

As mentioned in the previous Section, the GC is responsible for setting the properties of lines and basic (2D) shapes. GCs are therefore used with every Xlib drawing function. The `draw.c` (Section 17.3.1) program illustrates the setting of a variety of GC elements.

To create a GC use the (Xlib) function `XCreateGC()`. It has 4 parameters:

- the **Display** ID (pointer),
- the **Drawable** ID,
- **mask** (unsigned long) -- this controls how members of the following `XGCValues` structure may be set.
- a pointer to a `XGCValues` structure. This structure contains elements that can be set to change the values of an existing GC.

The `XGCValues` structure contains elements like `foreground`, `background`, `line_width`, `line_style`, *etc.* that we can set for obvious results. The mask has predefined values such as `GCForeground`, `GCBackground`, and `GCLineStyle`.

In `draw.c` we create a GC structure, `gc`, and set the foreground.

Xlib provides two macros `BlackPixel()` and `WhitePixel()` which will find the default black and white pixel values for a given `Display` and `Screen` if the default colourmaps are installed. Note that the reference to `BlackPixel()` and `WhitePixel()` can be a little confusing since the pixel colours returned may not necessarily be Black or White. `BlackPixel()` actually refers to the foreground colour and `WhitePixel()` refers to the background colour.

Therefore, to create a GC that *only* sets foreground colour to the default for a given display and screen:

```
gcv.foreground = BlackPixel(display, screen);
gc = XCreateGC(display, screen, GCForeground, &gcv);
```

where `gcv` is a `XGCValues` structure and `gc` a GC structure and `GCForeground` sets the mask to only allow alteration of the foreground.

To set *both* background and foreground:

```
gcv.foreground = BlackPixel(display, screen);
gcv.background = WhitePixel(display, screen);
gc = XCreateGC(display, screen,
               GCForeground | GCBackground, &gcv);
```

where we use the `|` (OR) in the mask parameter that allows both the values to be set in the `XGCValues` structure.

An alternative way to change GC elements is to use Xlib convenience functions to set appropriate GC values. Example functions include :

```
XSetForeground(), XSetBackground(), XSetLineAttributes() .
```

These set GC values for a given display and `gc`, for example:

```
XSetBackground(display, gc, WhitePixel(display, screen));
```

Further examples of their use are shown in the `draw.c` program (Section [17.3.1](#)).

## Two Dimensional Graphics

Xlib provides a whole range of 2D Graphics functions. See `draw.c` for examples in use. Most of these functions are fairly easy to understand and use. The functions basically draw a specific graphical primitive (point, line, polygon, arc, circle *etc.*) to a display according to a specific GC.

The simplest function is `XDrawPoint(Display *d, Drawable dr, GC gc, int x, int y)` which draws a point at position (x, y) to a given Drawable on a Display. This effectively colours a single pixel on the Display.

The function `XDrawPoints(Display *d, Drawable dr, GC gc, XPoint *pts, int n, int mode)` is similar except that an `n` element array of `XPoints` is drawn. The mode may be defined as being either `CoordModeOrigin` or `CoordModePrevious`. The former mode draws all points relative to the origin whilst the latter mode draws relative to the last point.

Other Xlib common drawing functions include:

`XDrawLine(Display *d, Drawable dr, GC gc, int x1, int y1, int x2, int y2)` draws a line between (x1, y1) and (x2, y2).

`XDrawLines(Display *d, Drawable dr, GC gc, XPoint *pts, int n, int mode)` draws a series of connected lines -- taking pairs of coordinates in the list. The mode is defined as for the `XDrawPoints()` function.

`XDrawRectangle(Display *d, Drawable dr, GC gc, int x, int y, int width, height)` draws a rectangle with top left hand corner coordinate (x, y) and of width and height.

`XFillRectangle(Display *d, Drawable dr, GC gc, int x, int y, int width, int height)` fills (shades interior) a rectangle. The `fill_style` controls what shading takes place.

`XFillPolygon(Display *d, Drawable dr, GC gc, XPoint *pts, int n, int mode, int mode)` fills a polygon whose outline is defined by the `*pts` array.

This function behaves much like `XDrawLines()`.

The shape parameter is either `Complex`, `Nonconvex` or `Convex` and controls how the server may configure the shading operation.

`XDrawArc(Display *d, Drawable dr, GC gc, int x, int y, int width, int height, int angle1, int angle2)` draws an arc.

`XFillArc(Display *d, Drawable dr, GC gc, int x, int y, int width, int height, int angle1, int angle2)` draws an arc and fills it.

The `x`, `y`, `width` and `height` define a *bounding* box for the arc. The arc is drawn (Fig. [16.1](#)) from the centre of the box. The `angle1` and `angle2` define the start and end points of the arc. The angles specify 1/64th degree steps measured anticlockwise. The `angle1` is relative to the 3 o'clock position and `angle2` is relative to `angle1`.

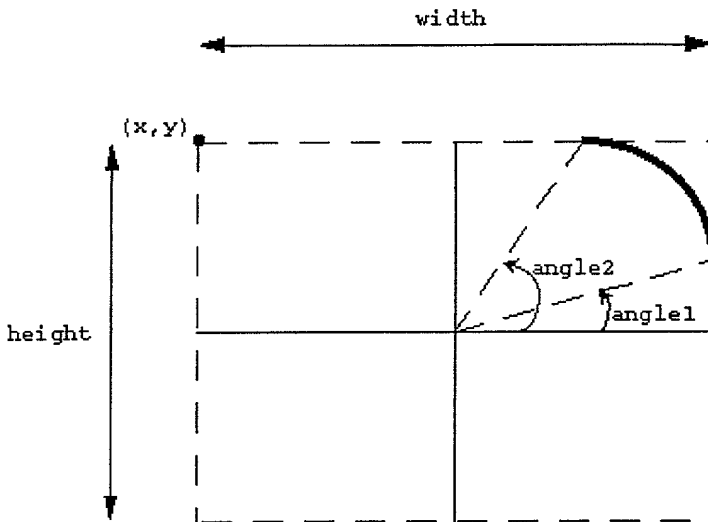


Fig.  arc.c display

Thus to draw a whole circle set the width and height equal to the diameter of the circle and  $\text{angle1} = 0$ ,  $\text{angle2} = 360 \times 64 = 23040$ .

## Pixmaps

If a window has been obscured then we will have to redraw the window when it gets re-exposed. This is the responsibility of the application and **not** the X window manager or system. In order to redraw a window we may have to go through all the drawing function calls that have previously been used to render our window's display. However, re-rendering a display in this manner will be cumbersome and may involve some complicated storage methods -- there maybe be many drawing functions involved and the order in which items are drawn may also be important

Fortunately, X provides a mechanism that overcomes these (and other less serious) problems. The use of Xlib **Pixmaps** is the best approach.

A *Pixmap* is an off-screen *Drawable* area.

We can draw to a *Pixmap* in the same way as we can draw to a *Window*. We use the standard Xlib graphics drawing functions (Section 16.3) but instead of specifying a window ID as the *Drawable* we provide a *Pixmap* ID. Note, however, that no immediate visual display effect will occur when drawing to a *Pixmap*. In order to see any effect we must *copy* the *Pixmap* to a *Window*.

The program `draw_input2.c` (Section 17.3.3) draws to *pixmap*s instead of to the window.

To create a *Pixmap* the `XCreatePixmap()` function should be used. This function takes 5 arguments and returns a *Pixmap* structure:

### Display\*

-- A pointer to the *Display* associated with *Pixmap*.

### Drawable

-- The screen on which to place Pixmap.

### Width, Height

-- The dimensions of the Pixmap.

### Depth

-- The number of bits of each *pixel*. Usually 8 by default. A Pixmap of depth 1 is usually called a *bitmap* and are used for icons and special *bitmap files*.

When you have finished using a Pixmap it is a good idea to *free* the memory in which it has been stored by calling:

```
XFreePixmap(Display*, Pixmap) .
```

If you want to clear a Pixmap (not done automatically) use `XFillRectangle()` to draw a background coloured rectangle that is the dimension of the whole Pixmap or use `XCclearArea()`, `XCclearWindow()` or similar functions.

To copy a Pixmap onto another Pixmap or a Window use:

```
XCopyArea(Display *display, Drawable source,
          Drawable destination, GC gc,
          int src_x, src_y,
          int width, int height,
          int dest_x, int dest_y);
```

where `(src_x, src_y)` specify the coordinates in the source pixmap where copy starts, width and height specify the dimensions of the copied area and `(dest_x, dest_y)` are the start coordinates in the destination where pixels are placed.

## Fonts

Fonts are necessary in Motif as all `XmString` are drawn to the screen using fonts residing in the X system. A *font* is a complete set of characters (upper-case and lower-case letters, punctuation marks and numerals) of one size and one typeface. In order for a Motif program to gain access to different typefaces, fonts must be loaded onto the X server. All X fonts are bitmapped.

Not all X servers support all fonts. Therefore it is best to check if a specific font has been loaded correctly within your Motif program. There is a standard X application program, *xlsfonts*, that lists the fonts available on a particular workstation.

Each font or character set name is referred to by a `String` name. Fonts are loaded into to an `Xlib Font` structure using the `XLoadFont()` function with a given `Display ID` and font name argument. The function returns a `Font` structure.

Another similar function is `XLoadQueryFont()` which takes the same arguments as above but returns an `XFontStruct` which contains the `Font` structure plus information describing the font.

An example function, `load_font()` which loads a font named `'fixed'`, which should be available on most systems but is still checked for is given below:

```
void load_font(XFontStruct **font_info)

{
    Display *display;
    char *fontname = "fixed";
    XFontStruct *font_info;

    display = XtDisplay(some_widget);

    /* load and get font info structure */

    if (( *font_info = XLoadQueryFont(display, fontname)) == NULL)
    { /* error - quit early */
        printf("Cannot load %s font\n", fontname);
        exit(1);
    }
}
```

Motif actually possesses its own font loading and setting functions. These include `XmFontListCreate()`, `XmFontListEntryCreate()`, `XmFontListEntryLoad()` and `XmFontListAdd()`. These are used in a similar fashion to the Xlib functions above except that they return an `XmFontList` structure. However, in this book, we will be only using fonts at the Xlib level and these Motif functions will not be considered further.

## XEvents

We should now be familiar with the basic notion of events in X and Motif. Mouse button presses, mouse motion and keyboard presses can be used to action menu, buttons *etc.*. These are all instances of events. Usually we are happy to let Motif take care of event scheduling with the `XtAppMainLoop()` function, and the setting of appropriate callback resources for widgets (Chapter 5).

Sometimes we may need to gain more control of events in X. To do this we will need to resort to Xlib. A specific example of this will be met in the Chapter 17 (the `draw_input1.c` program), where the default interaction, provided via callbacks in Motif, is inadequate for our required form of interaction.

## XEvent Types

There are many types of events in Xlib. A special `XEvent` structure is defined to take care of this. (See reference material [Mar96] for full details)

XEvents exist for all kinds of events, including: mouse button presses, mouse motions, key presses and events concerned with the window management. Most of the mouse/keyboard events are self explanatory and we have already studied them a little. Let us look at some window events further:

### XConfigureNotify

-- If a window changes size then this event is generated.

### XCirculateNotify

-- The event generated if the stacking order of windows has changed.

### XColormapNotify

-- The event generated if colormap changes are made.

### XCreateNotify, XDestroyNotify



-- The events generated when a window is created or deleted respectively.

### **XExpose, XNoExpose**

-- Windows can be stacked, moved around *etc.* If part of a window that has been previously obscured becomes visible again then it will need to be redrawn. An `XExpose` event is sent for this purpose. An `XExpose` event is also sent when a window first becomes visible.

**Note:** There is no guarantee that what has previously been drawn to the window will become immediately visible. In fact, it is totally up to the programmer to make sure that this happens by picking up an `XExpose` event (See Sections [16.4](#) and [17.3.3](#) on `Pixmap`s and `DrawingAreas`).

## **Writing Your Own Event Handler**

Most Motif applications will not need to do this since they can happily run within the standard application main loop event handling model. If you do need to resort to creating your own (Xlib) event handling routines, be warned: it can quickly become complex, involving a lot of Xlib programming.

Since, for the level of Motif programming described in this text, we will not need to resort to writing elaborate event handlers ourselves we will only study the basics of Motif/Xlib event handling and interaction.

The first step along this path is attaching a callback to an `XEvent` rather than a `Widget` callback action. From Motif (or Xt) you attach a callback to a particular event with the function `XtAddEventHandler()`, which takes 5 parameters:

### **Widget**

-- the ID of the widget concerned.

### **EventMask**

-- This can be used to allow the widget to be receptive to specific events. A complete list of event masks is given in the online support reference material. Multiple events can be assigned by ORing (`|`) masks together.

### **Nonmaskable**

-- A Boolean almost always set to `False`. If it is set to `True` then it can be activated on *nomaskable* events such as `ClientMessage`, `Graphics Expose`, `Mapping Notify`, `NoExpose`, `SelectionClear`, `SelectionNotify` or `SelectionRequest`.

### **Callback**

-- the callback function.

### **Client Data**

-- Additional data to be passed to the event handler.

As an example we could set an `expose_callback()` to be called by an `Expose` event by the following function call:

```
XtAddEventHandler(widget, ExposureMask, False,
                  expose_callback, NULL);
```

To set a callback, `motion_callback()`, that responds to left or middle mouse motion -- an event triggered when the mouse is moved whilst an appropriate mouse button is depressed -- we would write:

```
XtAddEventHandler(widget, Button1MotionMask | Button2MotionMask,
                  False, motion_callback, NULL);
```

There are two other steps that need to be done when writing our own event handler. These are:

- Intercepting Events, *and*
- Dispatching Events.

These two steps are basically what the `XtAppMainLoop()` takes care of in normal operation.

Two Xt functions are typically used in this context:

**XtAppNextEvent(XtAppContext, XEvent\*)**

gets the next event off the event *queue* for a given XtAppContext application.

**XtDispatchEvent(XEvent\*)**

dispatches the event so that callbacks can be invoked.

In between the retrieving of the next event and dispatching this event you may want to write some code that *intercepts* certain events.

Let us look at how the `XtAppMainLoop()` function is coded. Note the comments show where we may place custom application intercept code.

```
void XtAppMainLoop(XtAppContext app)
{
    XEvent event;
    for (;;) /* forever */
    {
        XtAppNextEvent(app, &event);

        /* Xevent read off queue */
        /* inspect structure and intercept perhaps? */

        /* intercept code would go here */

        XtDispatchEvent(&event);
    }
}
```

## Exercises

PLENTY OF SCOPE HERE

## The DrawingArea Widget

In this section we will look at how we create and use Motif's DrawingArea Widget which is concerned with the display of graphics. We will also put into practice the Xlib drawing and event scheduling routines met in Chapter 16.

## Creating a DrawingArea Widget

To create a DrawingArea Widget, use `XtVaCreateManagedWidget()` with `xmDrawingAreaWidgetClass` or use `XmCreateDrawingArea()`. Remember to include the `<Xm/DrawingA.h>` header file.

There is also a **DrawnButton** Widget which is a combination of a DrawingArea and a PushButton. There is a `xmDrawnButtonWidgetClass` and definitions are in the `<Xm/DrawnB.h>` header file.

## DrawingArea Resources and Callbacks

A DrawingArea will usually be placed inside a container widget --e.g. a Frame or a MainWindow -- and it is frequently *scrolled*. As such, it usually inherits size and other resources from its parent widget. You can, however, set resources like `XmNwidth` and `XmNheight` for the DrawingArea directly. The `XmNresizePolicy` resource may also need to be set to allow changes in dimension of the DrawingArea in a program. Possible values are:

```
XmRESIZE_ANY
    -- Allow any size of DrawingArea,
XmRESIZE_GROW
    -- Allow the DrawingArea to expand only from its initial size,
XmRESIZE_NONE
    -- No change in size is allowed.
```

Three callbacks are associated with this widget :

```
XmNexposeCallback
    -- This callback occurs when part of the window needs to be redrawn.
XmNresizeCallback
    -- If the dimensions of the window get changed this is called.
XmNinputCallback
    -- If input in the form of a mouse button or key press/release occurs this callback is invoked.
```

## Using DrawingAreas in Practice

We are now in a position to draw 2D graphics in Motif. Recall that all graphics drawing is performed at the Xlib level and so we have to attach the *higher level* motif widgets to the *lower level* Xlib structures (Chapter 16).

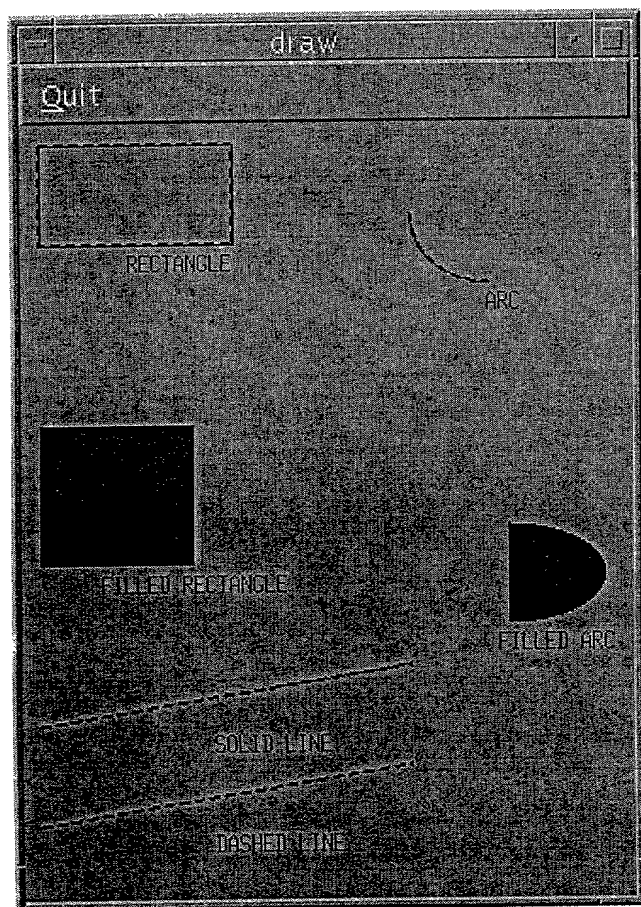
In order to draw anything in a DrawingArea Widget we basically need to do the following:

- Create a DrawingArea widget perhaps contained in other widgets.
- Obtain the Xlib `Display`, `window`, *etc.* ID's of the DrawingArea widget (Section 16.1).
- Draw, shade, *etc.* using Xlib graphics routines (Section 16.3).

## Basic Drawing -- draw.c

The draw.c program illustrates basic Motif/Xlib drawing principles:

- It creates a DrawingArea, draw, contained within a Main Window.
- We obtain Xlib Display and Screen IDs with `XtDisplay(draw)` and `XtScreen(draw)`.
- We create a Graphics Context with the foreground pixel set to the default `BlackPixel` of the Screen.
- We pass the Graphics Context value by making it the **user data** of the draw widget. To do this:
  1. Attach the appropriate data to the `XmUserData` resource. **Note:** this method can be used to pass any type of data not just graphics as illustrated here.
  2. To retrieve the data use `XtGetValues()`.
  3. The Xlib graphics routines are used to draw lines, rectangles, arcs and text to the DrawingArea (via the Display and Screen IDs).
- The font for the text labels for each drawing primitive is loaded and set in the function `load_font()`. The `font_height` is computed and used to determine where to place the text labels when drawn by `XDrawString()`.



**Fig. 17.1** Output of draw.c

The full program listing is:

```

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>

/* Prototype functions */

void quit_call(void);
void draw_cbk(Widget , XtPointer ,
              XmDrawingAreaCallbackStruct *);
void load_font(XFontStruct **);

/* XLIB Data */

Display *display;
Screen *screen_ptr;

main(int argc, char *argv[])
{
    Widget top_wid, main_w, menu_bar, draw, quit;
    XtAppContext app;
    XGCValues gcv;
    GC gc;

    top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
                               &argc, argv, NULL,
                               XmNwidth, 500,
                               XmNheight, 500,
                               NULL);

    main_w = XtVaCreateManagedWidget("main_window",
                                       xmMainWindowWidgetClass, top_wid,
                                       NULL);

    menu_bar = XmCreateMenuBar(main_w, "main_list",
                               NULL, 0);
    XtManageChild(menu_bar);

    /* create quit widget + callback */
    quit = XtVaCreateManagedWidget("Quit",
                                     xmCascadeButtonWidgetClass, menu_bar,
                                     XmNmnemonic, 'Q',
                                     NULL);

    XtAddCallback(quit, XmNactivateCallback, quit_call,
                  NULL);

    /* Create a DrawingArea widget. */
    draw = XtVaCreateWidget("draw",
                            xmDrawingAreaWidgetClass, main_w,
                            NULL);

    /* get XLib Display Screen and Window ID's for draw */
    display = XtDisplay(draw);
    screen_ptr = XtScreen(draw);

    /* set the DrawingArea as the "work area" of main window */
    XtVaSetValues(main_w,
                  XmNmenuBar, menu_bar,
                  XmNworkWindow, draw,

```

```

        NULL);

/* add callback for exposure event */
XtAddCallback(draw, XmNexposeCallback, draw_cbk, NULL);

/* Create a GC. Attach GC to the DrawingArea's
   XmNuserData.
   NOTE : This is a useful method to pass data */

gcv.foreground = BlackPixelOfScreen(screen_ptr);
gc = XCreateGC(display,
    RootWindowOfScreen(screen_ptr), GCForeground, &gcv);
XtVaSetValues(draw, XmNuserData, gc, NULL);

XtManageChild(draw);
XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

/* CALL BACKS */

void quit_call()
{
    printf("Quitting program\n");
    exit(0);
}

/* DrawingArea Callback. NOTE: cbk->reason says type of
   callback event */

void
draw_cbk(Widget w, XtPointer data,
    XmDrawingAreaCallbackStruct *cbk)
{
    char str1[25];
    int len1, width1, font_height;
    unsigned int width, height;
    int x, y, angle1, angle2, x_end, y_end;
    unsigned int line_width = 1;
    int line_style = LineSolid;
    int cap_style = CapRound;
    int join_style = JoinRound;
    XFontStruct *font_info;
    XEvent *event = cbk->event;
    GC gc;
    Window win = XtWindow(w);

    if (cbk->reason != XmCR_EXPOSE)
    {
        /* Should NEVER HAPPEN for this program */
        printf("X is screwed up!!\n");
        exit(0);
    }

    /* get font info */

    load_font(&font_info);

    font_height = font_info->ascent + font_info->descent;

    /* get gc from Drawing Area user data */

    XtVaGetValues(w, XmNuserData, &gc, NULL);

```

[illegible]

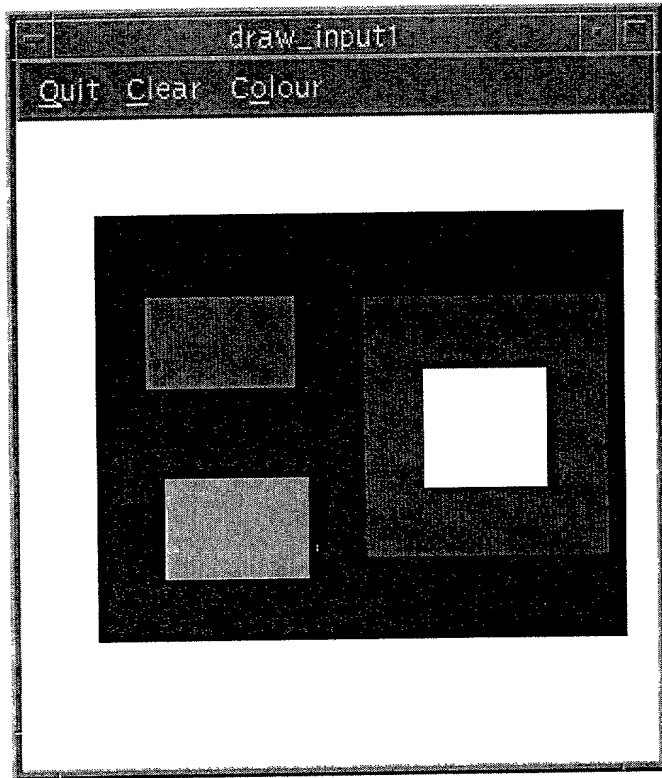
[illegible]

The previous program only illustrated one aspect of the DrawingArea widget, *i.e.* displaying graphics. Another important aspect of this widget is how the widget accepts input. In this Section we will develop a program that illustrate how input is processed in the DrawingArea. We will write a program,



`draw_input1.c`, that highlights some deficiencies in the default event handling capabilities within a practical application.

The `draw_input1.c` program accepts mouse input in the `DrawingArea`. It allows the user to select a colour (as we have seen previously) and then draw a variable size rectangle that is shaded with the chosen colour. A clear `DrawingArea` facility is also provided.



**Fig. 17.2** Output of `draw_input1.c`

In order to achieve a practical and intuitive manner of user interaction we will need to detect 3 different mouse events (all events described below refer to the *left* mouse button):

- The mouse button **down** indicates that input is about to start and the coordinates selected here are the *top left* corner of the rectangle.
- The mouse button **up** indicates the end of input and a rectangle is drawn using the coordinates selected here -- *bottom right* corner.
- It is useful to provide visual feedback as to the size and location of the rectangle as the user is moving the mouse. We can do this by detecting a mouse **motion** and drawing a silhouette of the rectangle as the mouse moves. (See Fig. 17.3). **Note:** This is common practice in many applications where selection of several items, as well a drawing, requires such outlines to be drawn.

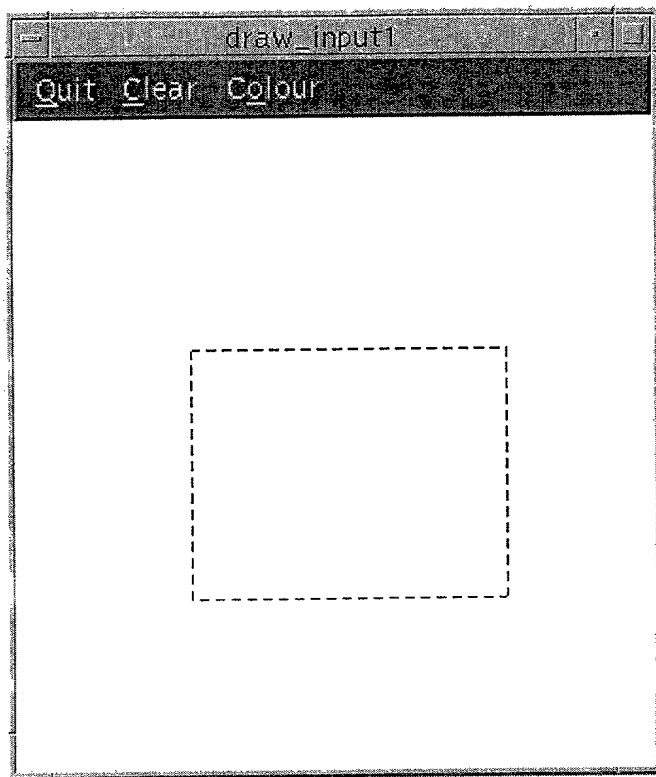


Fig. 17.3 Silhouette outline of rectangle during input ( draw\_input1.c)

To detect mouse clicks *up* and *down* we can use the `XmNinputCallback` resource. However, the default setting of callback resources in a `DrawingArea` Widget does not allow for mouse motion to be detected as we would like. We therefore have to override the default callback options.

Every Widget has a **Translation Table** (Section 5.8.2) that contains a list of events that it can receive and actions that it is to take upon receipt of an event. We basically have to create a new translation table to achieve our desired interaction described above.

We have already defined the translation table format in Section 5.8.2. A translation table consists of events like the below excerpt of the default `DrawingArea` translation:

```
.....
<Btn1Down>:   DrawingAreaInput() ManagerGadgetArm()
<Btn1Up>:     DrawingAreaInput() ManagerGadgetActivate()
<Btn1Motion>: ManagerGadgetButtonMotion()
.....
```

Our particular problem is that button motion does not get passed to the `DrawingAreaInput()` function that notifies the program of an input event.

To create a new translation table, for our purpose, we simply include the functions and events we need. In this case:

```
<Btn1Down>:   draw_cbk(down) ManagerGadgetArm()
<Btn1Up>:     draw_cbk(up) ManagerGadgetActivate()
<Btn1Motion>: draw_cbk(motion) ManagerGadgetButtonMotion()
```

where `draw_cbk()` is our callback that performs the drawing. We use the same callback to detect each mouse button down, up or motion action. This is achieved by sending a message to the callback that

identifies each action. The arm, activate and motion gadget manager functions control the (default) display of an event action.

In a motif program, we set up our translation table in a `String` structure and use the `XtParseTranslationTable(String*)` to attach a translation table to the `XmNtranslations` resource. We **must** also register the callback with the actions associated with the translation events. We use the `XtAppAddActions()` function to do this.

For the above example we create the `String` as follows:

```
String translations =
"<Btn1Motion>: draw_cbk(motion)
                ManagerGadgetButtonMotion() \n\
<Btn1Down>: draw_cbk(down) ManagerGadgetArm() \n\
<Btn1Up>: draw_cbk(up) ManagerGadgetActivate()";
```

and register the callback and create a `DrawingArea` widget with the correct actions with the following code:

```
actions.string = "draw_cbk";
actions.proc = draw_cbk;
XtAppAddActions(app, &actions, 1);

draw = XtVaCreateWidget("draw",
xmDrawingAreaWidgetClass, main_w,
XmNtranslations, XtParseTranslationTable(translations),
XmNbackground, WhitePixelOfScreen(XtScreen(main_w)),
NULL);
```

The Callback function would be prototyped by:

```
draw_cbk(Widget w, XButtonEvent *event, String *args, int *num_args).
```

On calling the function, we simply inspect the `args[0]` `String` to see if an up, down or motion event has occurred and take the appropriate actions described below:

### Mouse Down

-- Simply remember the (x,y) coordinates of the mouse. These are found in the `x, y` elements of the event structure.

### Mouse Motion

-- Draw a *dashed* line silhouette of the box whose current size is determined by mouse down (x,y) and current mouse position. **Note:** We set the Graphics Context Logical Function to `GXinvert` which means that pixels simply get inverted. We must invert them once more to get them back to their original state before we redraw again at another mouse position.

### Mouse Up

-- We finally draw our rectangle with the desired colour.

The complete program listing of `draw_input1.c` is :

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
```

```

#include <Xm/DrawingA.h>

/* Prototype callbacks */

void quit_call(void);
void clear_call(void);
void colour_call(Widget , int);
void draw_cbk(Widget , XButtonEvent *,
              String *, int *);

GC gc;
XGCValues gcv;
Widget draw;
String colours[] = { "Black", "Red", "Green", "Blue",
                    "Grey", "White"};
long int fill_pixel = 1; /* stores current colour
                        of fill - black default */
Display *display; /* xlib id of display */
Colormap cmap;

main(int argc, char *argv[])

{
    Widget top_wid, main_w, menu_bar, quit, clear, colour;
    XtAppContext app;
    XmString quits, clears, colourss, red, green,
              blue, black, grey, white;
    XtActionsRec actions;

    String translations =
"<Btn1Motion>: draw_cbk(motion)
                ManagerGadgetButtonMotion() \n\
<Btn1Down>: draw_cbk(down) ManagerGadgetArm() \n\
<Btn1Up>: draw_cbk(up) ManagerGadgetActivate() ";

    top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
                              &argc, argv, NULL,
                              XmNwidth, 500,
                              XmNheight, 500,
                              NULL);

    main_w = XtVaCreateManagedWidget("main_window",
                                     xmMainWindowWidgetClass, top_wid,
                                     XmNwidth, 500,
                                     XmNheight, 500,
                                     NULL);

    /* Create a simple MenuBar that contains three menus */
    quits = XmStringCreateLocalized("Quit");
    clears = XmStringCreateLocalized("Clear");
    colourss = XmStringCreateLocalized("Colour");

    menu_bar = XmVaCreateSimpleMenuBar(main_w, "main_list",
                                       XmVaCASCADEBUTTON, quits, 'Q',
                                       XmVaCASCADEBUTTON, clears, 'C',
                                       XmVaCASCADEBUTTON, colourss, 'o',
                                       NULL);

    XtManageChild(menu_bar);

    /* First menu is quit menu -- callback is quit_call() */

    XmVaCreateSimplePulldownMenu(menu_bar, "quit_menu", 0,
                                quit_call, XmVaPUSHBUTTON, quits, 'Q', NULL, NULL,

```

```

        NULL);
    XmStringFree(quits);

/* Second menu is clear menu -- callback is clear_call() */

    XmVaCreateSimplePulldownMenu(menu_bar, "clear_menu", 1,
        clear_call, XmVaPUSHBUTTON, clears, 'C', NULL, NULL,
        NULL);
    XmStringFree(clears);

    /* create colour pull down menu */

    black = XmStringCreateLocalized(colours[0]);
    red = XmStringCreateLocalized(colours[1]);
    green = XmStringCreateLocalized(colours[2]);
    blue = XmStringCreateLocalized(colours[3]);
    grey = XmStringCreateLocalized(colours[4]);
    white = XmStringCreateLocalized(colours[5]);

    colour = XmVaCreateSimplePulldownMenu(menu_bar,
        "edit_menu", 2, colour_call,
        XmVaRADIOBUTTON, black, 'k', NULL, NULL,
        XmVaRADIOBUTTON, red, 'R', NULL, NULL,
        XmVaRADIOBUTTON, green, 'G', NULL, NULL,
        XmVaRADIOBUTTON, blue, 'B', NULL, NULL,
        XmVaRADIOBUTTON, grey, 'e', NULL, NULL,
        XmVaRADIOBUTTON, white, 'W', NULL, NULL,
        XmNradioBehavior, True,
        /* RowColumn resources to enforce */
        XmNradioAlwaysOne, True,
        /* radio behavior in Menu */
        NULL);

    XmStringFree(black);
    XmStringFree(red);
    XmStringFree(green);
    XmStringFree(blue);
    XmStringFree(grey);
    XmStringFree(white);

/* Create a DrawingArea widget. */
/* make new actions */

    actions.string = "draw_cbk";
    actions.proc = draw_cbk;
    XtAppAddActions(app, &actions, 1);

    draw = XtVaCreateWidget("draw",
        xmDrawingAreaWidgetClass, main_w,
        XmNtranslations, XtParseTranslationTable(translations),
        XmNbackground, WhitePixelOfScreen(XtScreen(main_w)),
        NULL);

    cmap = DefaultColormapOfScreen(XtScreen(draw));
    display = XtDisplay(draw);

/* set the DrawingArea as the "work area" of main window */
    XtVaSetValues(main_w,
        XmNmenuBar, menu_bar,
        XmNworkWindow, draw,
        NULL);

```

```

/* Create a GC. Attach GC to DrawingArea's XmNuserData. */
gcv.foreground = BlackPixelOfScreen(XtScreen(draw));
gc = XCreateGC(XtDisplay(draw),
               RootWindowOfScreen(XtScreen(draw)),
               GCForeground, &gcv);

XtManageChild(draw);
XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

/* CALL BACKS */

void quit_call()
{
    printf("Quitting program\n");
    exit(0);
}

void clear_call() /* clear work area */
{
    XClearWindow(display, XtWindow(draw));
}

/* called from any of the "Colour" menu items.
   Change the colour of the label widget.
   Note: we have to use dynamic setting with setargs()..
   */
void
colour_call(Widget w, int item_no)
/* w -- menu item that was selected */
/* item_no --- the index into the menu */
{
    int n = 0;
    Arg args[1];

    XColor xcolour, spare; /* xlib colour struct */

    if (XAllocNamedColor(display, cmap, colours[item_no],
                        &xcolour, &spare) == 0)
        return;

    /* remember new colour */
    fill_pixel = xcolour.pixel;
}

/* DrawingArea Callback.*/

void draw_cbk(Widget w, XButtonEvent *event,
              String *args, int *num_args)
{
    static Position x, y, last_x, last_y;
    Position width, height;

    int line_style;
    unsigned int line_width = 1;
    int cap_style = CapRound;
    int join_style = JoinRound;

    if (strcmp(args[0], "down") == 0)
        { /* anchor initial point (save its value) */

```

```

        x = event->x;
        y = event->y;
    }
else
    if (strcmp(args[0], "motion") == 0)
    { /* draw "ghost" box to show where it could go */
        /* undraw last box */

        line_style = LineOnOffDash;

        /* set line attributes */

        XSetLineAttributes(event->display, gc,
            line_width, line_style, cap_style, join_style);

        gcv.foreground
            = WhitePixelOfScreen(XtScreen(w));

        XSetForeground(event->display, gc,
            gcv.foreground);

        XSetFunction(event->display, gc, GXinvert);

        XDrawLine(event->display, event->window, gc,
            x, y, last_x, y);
        XDrawLine(event->display, event->window, gc,
            last_x, y, last_x, last_y);
        XDrawLine(event->display, event->window, gc,
            last_x, last_y, x, last_y);
        XDrawLine(event->display, event->window, gc,
            x, last_y, x, y);

        /* Draw New Box */
        gcv.foreground
            = BlackPixelOfScreen(XtScreen(w));
        XSetForeground(event->display, gc,
            gcv.foreground);

        XDrawLine(event->display, event->window, gc,
            x, y, event->x, y);
        XDrawLine(event->display, event->window, gc,
            event->x, y, event->x, event->y);
        XDrawLine(event->display, event->window, gc,
            event->x, event->y, x, event->y);
        XDrawLine(event->display, event->window, gc,
            x, event->y, x, y);
    }
else
    if (strcmp(args[0], "up") == 0)
    { /* draw full line */

        XSetFunction(event->display, gc, GXcopy);

        line_style = LineSolid;

        /* set line attributes */

        XSetLineAttributes(event->display, gc,
line_width, line_style, cap_style, join_style);

        XSetForeground(event->display, gc, fill_pixel);

        XDrawLine(event->display, event->window, gc,

```

```

        x, y, event->x, y);
XDrawLine(event->display, event->window, gc,
          event->x, y, event->x, event->y);
XDrawLine(event->display, event->window, gc,
          event->x, event->y, x, event->y);
XDrawLine(event->display, event->window, gc,
          x, event->y, x, y);

width = event->x - x;
height = event->y - y;
XFillRectangle(event->display, event->window,
               gc, x, y, width, height);
}
last_x = event->x;
last_y = event->y;
}

```

## Drawing to a pixmap -- draw\_input2.c

One problem the draw\_input1.c program has is that if the window was covered and then exposed the picture would not *redraw* itself (Section 16.6). To see this for yourself run the draw\_input1.c obscure the MainWindow with another window and then click on the draw\_input1.c frame to bring it to the foreground and note the appearance of the window.

Indeed, the best method to store the picture we draw is via a Pixmap. Since the drawing in this application is an interactive process it would be very difficult to redraw the picture unless we used **Pixmaps**. There is no way that we could predict how the user would use the application and storing each drawing stroke would become complex. The best approach is to store the drawing data in a Pixmap by writing directly to a Pixmap. Obtaining an immediate visual feedback is also important (the user needs to see what he has drawn) so we also draw directly to the display when data is being input. When an expose event is detected all we need to do is remap the pixmap to the window.

The draw\_input2.c program does exactly the same task as draw\_input1.c but draws to a pixmap which can be remapped to the window upon an exposure.

The major differences between the programs are:

- A Pixmap is created with the XCreatePixmap() function. The Pixmap is made the same size as the DrawingArea and is assigned to default Screen and DepthOfScreen values.
- We still draw some things to the DrawingArea only. The *dashed* boxes which only serve to indicate the current rectangle size **do not** need to be permanently stored and are unlikely to be drawn, covered and re-exposed in between another mouse motion or the mouse up event.
- We draw the filled colour rectangle to **both** DrawingArea and Pixmap so that we get an immediate effect of drawing and we have the Pixmap backup store.
- Upon an expose event we use XCopyArea() to copy the Pixmap to the DrawingArea window.

The draw\_input2.c program listing is as follows:

```

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>

```



```

/* Prototype callbacks */

void quit_call(void);
void clear_call(void);
void colour_call(Widget , int);
void draw_cbk(Widget , XButtonEvent *, String *, int *);
void expose(Widget , XtPointer ,
            XmDrawingAreaCallbackStruct *);

GC gc;
XGCValues gcv;
Widget draw;
Display *display; /* xlib id of display */
Screen *screen;
Colormap cmap;
Pixmap pix;
Dimension width, height; /* store size of pixmap */
String colours[] = { "Black", "Red", "Green", "Blue",
                    "Grey", "White"};
long int fill_pixel = 1; /* stores current colour of fill
                        - black default */

main(int argc, char *argv[])
{
    Widget top_wid, main_w, menu_bar, quit, clear, colour;
    XtAppContext app;
    XmString quits, clears, colourss, red, green, blue,
        black, grey, white;
    XtActionsRec actions;

    String translations =
        "<Btn1Motion>: draw_cbk(motion) ManagerGadgetButtonMotion() \n\
        <Btn1Down>: draw_cbk(down) ManagerGadgetArm() \n\
        <Btn1Up>: draw_cbk(up) ManagerGadgetActivate()";

    top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
        &argc, argv, NULL,
        NULL);

    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass, top_wid,
        NULL);

    /* Create a simple MenuBar that contains three menus */
    quits = XmStringCreateLocalized("Quit");
    clears = XmStringCreateLocalized("Clear");
    colourss = XmStringCreateLocalized("Colour");

    menu_bar = XmVaCreateSimpleMenuBar(main_w, "main_list",
        XmVaCASCADEBUTTON, quits, 'Q',
        XmVaCASCADEBUTTON, clears, 'C',
        XmVaCASCADEBUTTON, colourss, 'o',
        NULL);

    XtManageChild(menu_bar);

    /* First menu is the quit menu
       -- callback is quit_call() */

    XmVaCreateSimplePulldownMenu(menu_bar, "quit_menu", 0,
        quit_call,
        XmVaPUSHBUTTON, quits, 'Q', NULL, NULL,
        NULL);

```

**Abstract**

```

/* get pixmap of DrawingArea */
XtVaGetValues(draw, XmNwidth, &width, XmNheight, &height, NULL);

pix = XCreatePixmap(display, RootWindowOfScreen(screen),
                    width, height, DefaultDepthOfScreen(screen));

/* initial white pixmap */
XSetForeground(XtDisplay(draw), gc,
               WhitePixelOfScreen(XtScreen(draw)));

XFillRectangle(display, pix, gc, 0, 0, width, height);

/* reset gc with current colour */
XSetForeground(display, gc, fill_pixel);

/* set the DrawingArea as the "work area" of the main window */
XtVaSetValues(main_w,
              XmNmenuBar, menu_bar,
              XmNworkWindow, draw,
              NULL);

/* add callback for exposure event */
XtAddCallback(draw, XmNexposeCallback, expose, NULL);

XtManageChild(draw);
XtRealizeWidget(top_wid);
XtAppMainLoop(app);

CALL BACKS */

d quit_call()

printf("Quitting program\n");
exit(0);

d clear_call(Widget w, int item_no) /* clear work area */

* clear pixmap with white */
XSetForeground(XtDisplay(draw), gc,
               WhitePixelOfScreen(XtScreen(draw)));

XFillRectangle(display, pix, gc, 0, 0, width, height);

/* reset gc with current colour */
XSetForeground(display, gc, fill_pixel);

/* copy pixmap to window of drawing area */
XCopyArea(display, pix, XtWindow(draw), gc,
           0, 0, width, height, 0, 0);

expose is called whenever all or portions of the drawing area is
exposed.

d
expose(Widget draw, XtPointer client_data,
       XmDrawingAreaCallbackStruct *cbk)

XCopyArea(cbk->event->xexpose.display, pix, cbk->window, gc,
           0, 0, width, height, 0, 0);

```

```

/* called from any of the "Colour" menu items.
   Change the colour of the label widget.
   Note: we have to use dynamic setting with setargs().
*/

void
colour_call(Widget w, int item_no)

/* w = menu item that was selected
   item_no = the index into the menu
*/

{
    int n = 0;
    Arg args[1];

    XColor xcolour, spare; /* xlib color struct */

    if (XAllocNamedColor(display, cmap, colours[item_no],
                        &xcolour, &spare) == 0)
        return;

    /* remember new colour */
    fill_pixel = xcolour.pixel;
}

/* DrawingArea Callback */

void draw_cbk(Widget w, XButtonEvent *event,
              String *args, int *num_args)
{
    static Position x, y, last_x, last_y;
    Position width, height;

    int line_style;
    unsigned int line_width = 1;
    int cap_style = CapRound;
    int join_style = JoinRound;

    if (strcmp(args[0], "down") == 0)
    { /* anchor initial point (i.e., save its value) */
        x = event->x;
        y = event->y;
    }
    else
    if (strcmp(args[0], "motion") == 0)
    { /* draw "ghost" box to show where it could go */

        /* undraw last box */

        line_style = LineOnOffDash;

        /* set line attributes */

        XSetLineAttributes(event->display, gc, line_width,
                           line_style, cap_style, join_style);
    }
}

```

```

gcv.foreground = WhitePixelOfScreen(XtScreen(w));
XSetForeground(event->display, gc, gcv.foreground);

XSetFunction(event->display, gc, GXinvert);

XDrawLine(event->display, event->window, gc,
           x, y, last_x, y);
XDrawLine(event->display, event->window, gc,
           last_x, y, last_x, last_y);
XDrawLine(event->display, event->window, gc,
           last_x, last_y, x, last_y);
XDrawLine(event->display, event->window, gc,
           x, last_y, x, y);

/* Draw New Box */

gcv.foreground = BlackPixelOfScreen(XtScreen(w));
XSetForeground(event->display, gc, gcv.foreground);

XDrawLine(event->display, event->window, gc,
           x, y, event->x, y);
XDrawLine(event->display, event->window, gc,
           event->x, y, event->x, event->y);
XDrawLine(event->display, event->window, gc,
           event->x, event->y, x, event->y);
XDrawLine(event->display, event->window, gc,
           x, event->y, x, y);
}
else
if (strcmp(args[0], "up") == 0)
{ /* draw full line; get GC and use in XDrawLine() */

XSetFunction(event->display, gc, GXcopy);

line_style = LineSolid;

/* set line attributes */

XSetLineAttributes(event->display, gc,
                   line_width, line_style, cap_style, join_style);

XSetForeground(event->display, gc, fill_pixel);

XDrawLine(event->display, event->window, gc,
           x, y, event->x, y);
XDrawLine(event->display, event->window, gc,
           event->x, y, event->x, event->y);
XDrawLine(event->display, event->window, gc,
           event->x, event->y, x, event->y);
XDrawLine(event->display, event->window, gc,
           x, event->y, x, y);

width = event->x - x;
height = event->y - y;
XFillRectangle(event->display, event->window, gc,
               x, y, width, height);

/* only need to draw final selection to pixmap */

XDrawLine(event->display, pix, gc,
           x, y, event->x, y);
XDrawLine(event->display, pix, gc,

```

```

        event->x, y, event->x, event->y);
XDrawLine(event->display, pix, gc,
        event->x, event->y, x, event->y);
XDrawLine(event->display, pix, gc,
        x, event->y, x, y);

XFillRectangle(event->display, pix, gc,
        x, y, width, height);
}

last_x = event->x;
last_y = event->y;
}

```

## Exercises

NEED SOME EXERCISE -- RUN

## References

### Add93a

Addison-Wesley Developers Press, One Jacob Way, Reading, MA 01867, USA.  
*Desktop KornShell Graphical Programming*, 1993.

### Add93b

Addison-Wesley Developers Press,, One Jacob Way Reading, MA 01867, USA.  
*Desktop KornShell User's Guide*, 1993.

### Add94a

Addison-Wesley Developers Press, One Jacob Way Reading, MA 01867, USA.  
*Common Desktop Environment Advanced User's and System Administrator's Guide*, 1994.

### Add94b

Addison-Wesley Developers Press, One Jacob Way Reading, MA 01867, USA.  
*Common Desktop Environment Application Builder User's Guide*, 1994.

### Add94c

Addison-Wesley Developers Press, One Jacob Way Reading, MA 01867, USA.  
*Common Desktop Environment Help System Author's and Programmer's Guide*, 1994.

### Add94d

Addison-Wesley Developers Press, One Jacob Way Reading, MA 01867, USA.  
*Common Desktop Environment Help System Author's and Programmer's Guide*, 1994.

### Add94e

Addison-Wesley Developers Press, One Jacob Way Reading, MA 01867, USA.  
*Common Desktop Environment Programmer's Guide*, 1994.

### Add94f

Addison-Wesley Developers Press, One Jacob Way Reading, MA 01867, USA.  
*Common Desktop Environment Programmer's Overview*, 1994.

**Add94g**

Addison-Wesley Developers Press, One Jacob Way Reading, MA 01867, USA.  
*Common Desktop Environment User's Guide*, 1994.

**CDO92**

E. Cutler, Gilly D., and T. O'Reilly.  
*The X Window System in a Nutshell*.  
O'Reilly & Associates, Sebastopol, CA, USA, 2 edition, 1992.

**Cul94**

Culwin.  
*An X/Motif Programmers Primer*.  
Prentice Hall, London, UK, 1994.

**FE92**

D. Flanagan (Ed.).  
*Volume Five: X Toolkit Intrinsic Reference Manual*.  
O'Reilly & Associates, Sebastopol, CA, USA, 1992.

**Fla91**

D. Flanagan.  
*Programmer's Supplement for R5 of the X Window System*.  
O'Reilly & Associates, Sebastopol, CA, USA, 1991.

**Gas92**

T. Gaskins.  
*PHIGS Programming Manual: 3D Programming in X*.  
O'Reilly & Associates, Sebastopol, CA, USA, 1992.

**Hel94a**

D. Heller.  
*Volume Six A: Motif 1.2 Programming Manual*.  
O'Reilly & Associates, Sebastopol, CA, USA, 1994.

**Hel94b**

D. Heller.  
*Volume Six B: Motif 1.2 Reference Manual*.  
O'Reilly & Associates, Sebastopol, CA, USA, 1994.

**JK94**

E.F. Johnson and Reichard K.  
*Power Programming: Motif*.  
O'Reilly & Associates, New York, USA, 2 edition, 1994.

**KE92**

L. Kosko (Ed.).  
*PHIGS Reference Manual: 3D Programming in X*.  
O'Reilly & Associates, Sebastopol, CA, USA, 1992.

**Mar96**

A.D. Marshall.

*The Road to X/Motif Programming: Online Reference Material.*

Thomson International, London, U.K., 1996.

**MP92**

L. Mui and E. Pearce.

*Volume Eight: X Window System Administrator's Guide.*

O'Reilly & Associates, Sebastopol, CA, USA, 1992.

**NE92a**

A. Nye (Ed.).

*Volume 0: X Protocol Reference Manual.*

O'Reilly & Associates, Sebastopol, CA, USA, 3 edition, 1992.

**NE92b**

A. Nye (Ed.).

*Volume Two: Xlib Reference Manual.*

O'Reilly & Associates, Sebastopol, CA, USA, 3 edition, 1992.

**New92**

J. Newmarch.

*The X Window System and Motif: A Fast Track Approach.*

Addison Wesley, New York, USA, 1992.

**NO92**

A. Nye and T. O'Reilly.

*Volume Four: X Toolkit Intrinsics Programming Manual (Motif Edition).*

O'Reilly & Associates, Sebastopol, CA, USA, 1992.

**Nye92**

A. Nye.

*Volume One: Xlib Programming Manual.*

O'Reilly & Associates, Sebastopol, CA, USA, 3 edition, 1992.

**Ope93**

Open Software Foundation, London, UK.

*OSF/Motif Style Guide*, 1993.

**Ope95a**

Open Software Foundation, London, UK.

*OSF/Motif 2.0 Programming Manual*, 1995.

**Ope95b**

Open Software Foundation, London, UK.

*OSF/Motif 2.0 Reference Manual*, 1995.

**Ope95c**

Open Software Foundation, London, UK.

*OSF/Motif Widget Writer's Guide*, 1995.

**QO90**

V. Quercia and T. O'Reilly.



*Volume Three: X Window System User's Guide.*  
O'Reilly & Associates, Sebastopol, CA, USA, 1990.

**QO91**

V. Quercia and T. O'Reilly.  
*Volume Three: X Window System User's Guide (Motif Edition).*  
O'Reilly & Associates, Sebastopol, CA, USA, 1991.

**Ros93**

R.K Rost.  
*X and Motif Quick Reference Guide.*  
Digital Press, New York, USA, 2 edition, 1993.

**RR92**

L. Reiss and J. Radin.  
*X Window: Inside and Out.*  
McGraw Hill, New York, USA, 1992.

## About this document ...

### X Window/Motif Programming

This document was generated using the LaTeX2HTML translator Version 97.1 (release) (July 13th, 1997)

Copyright © 1993, 1994, 1995, 1996, 1997, Nikos Drakos, Computer Based Learning Unit, University of Leeds.

The command line arguments were:

**latex2html** -split 1 -address dave@cs.cf.ac.uk X\_book\_caller.

The translation was initiated by Dave Marshall on 1/4/1999

---

<a href="#">Next</a>	<a href="#">Up</a>	<a href="#">Previous</a>
----------------------	--------------------	--------------------------

[dave@cs.cf.ac.uk](mailto:dave@cs.cf.ac.uk)

000260" / 0023 / 950

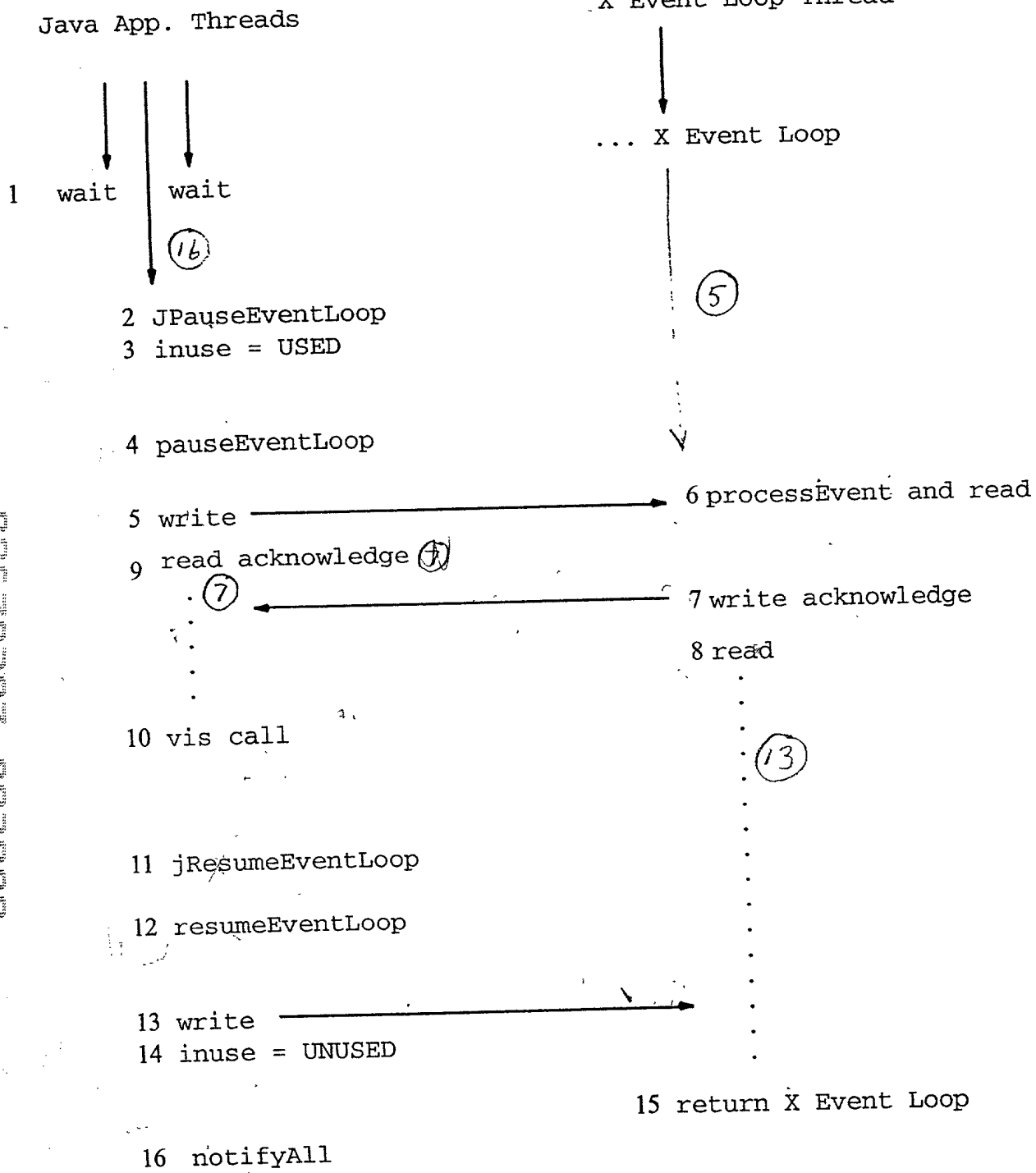


FIG. 1

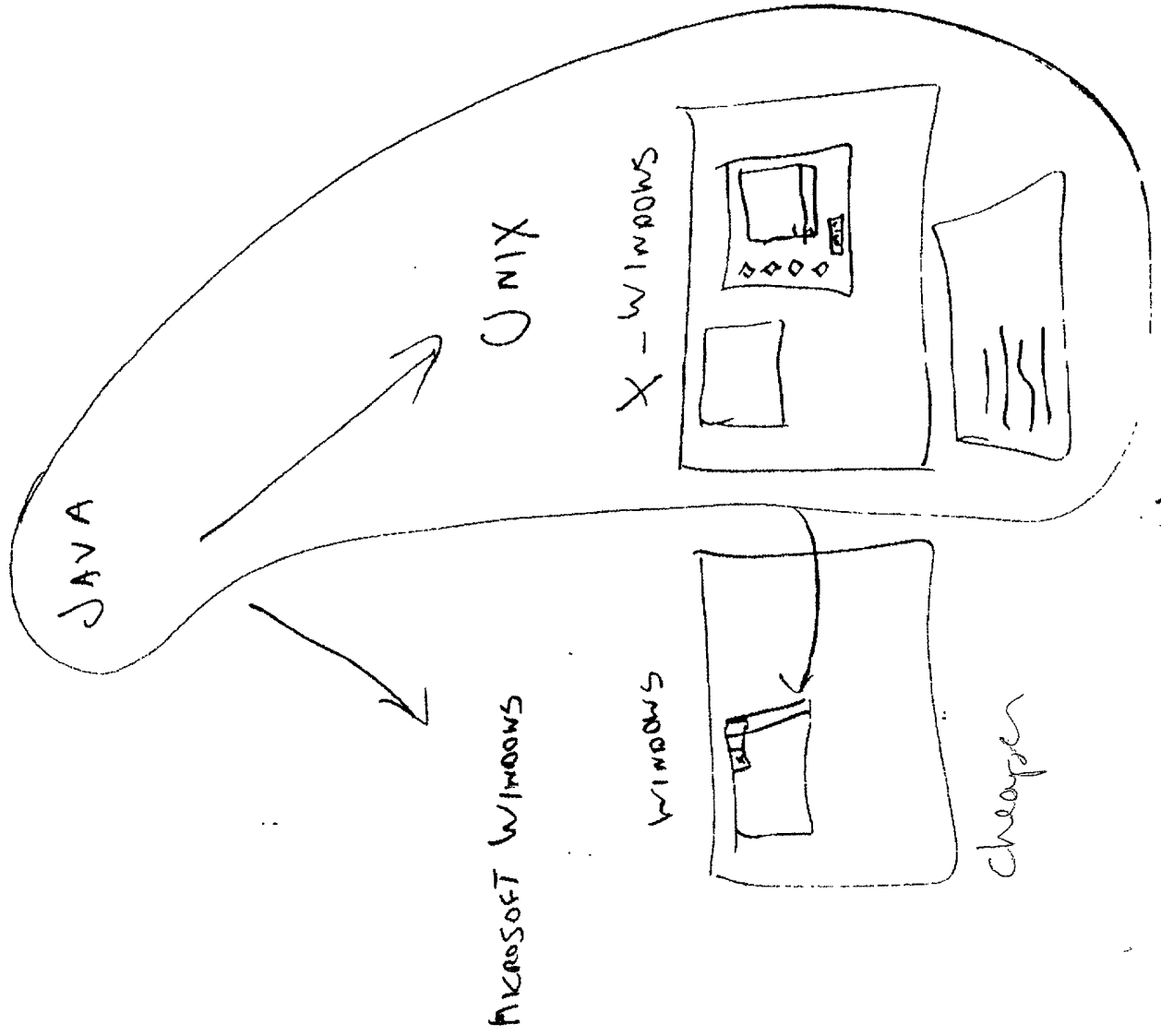


FIG. 4

ATHENA	NOTICE
XT	X W
X-WINDOWS	

1000-5 1000-5 1000-5

Problem to be solved

Score 1

## Processes (OS running multiple programs)

# Program Counter

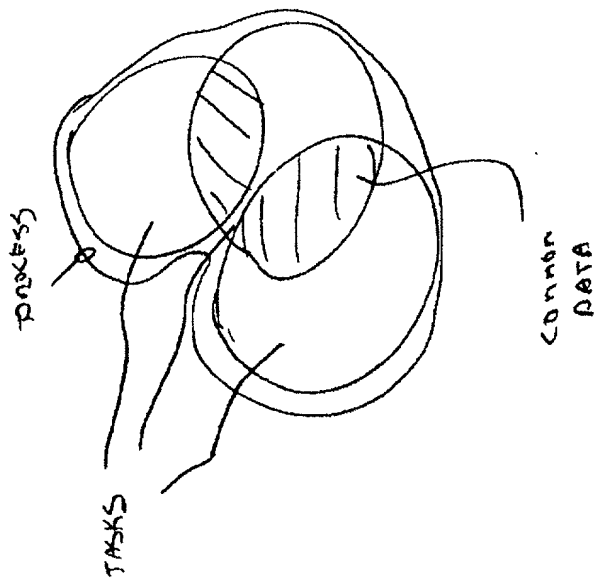
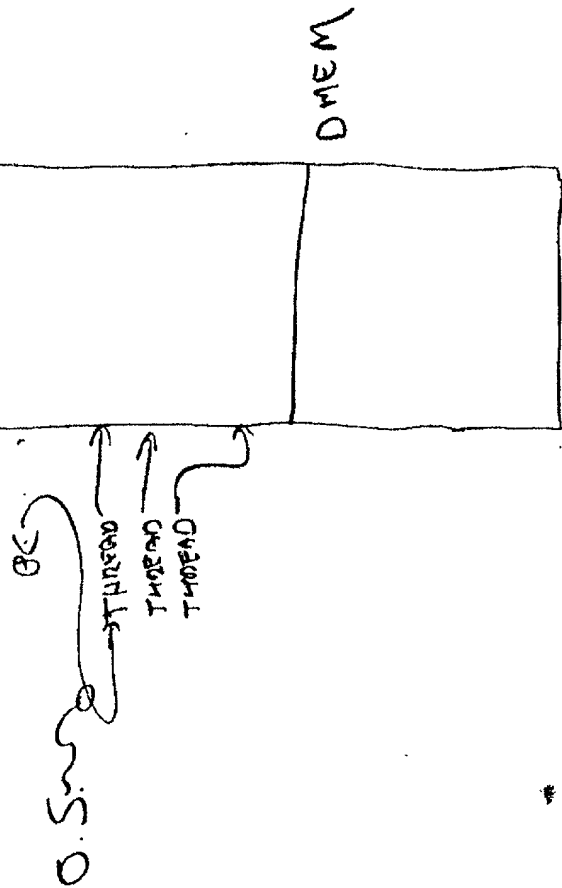


FIG. 5

## Multithreads in JAVA for programming efficiency

②

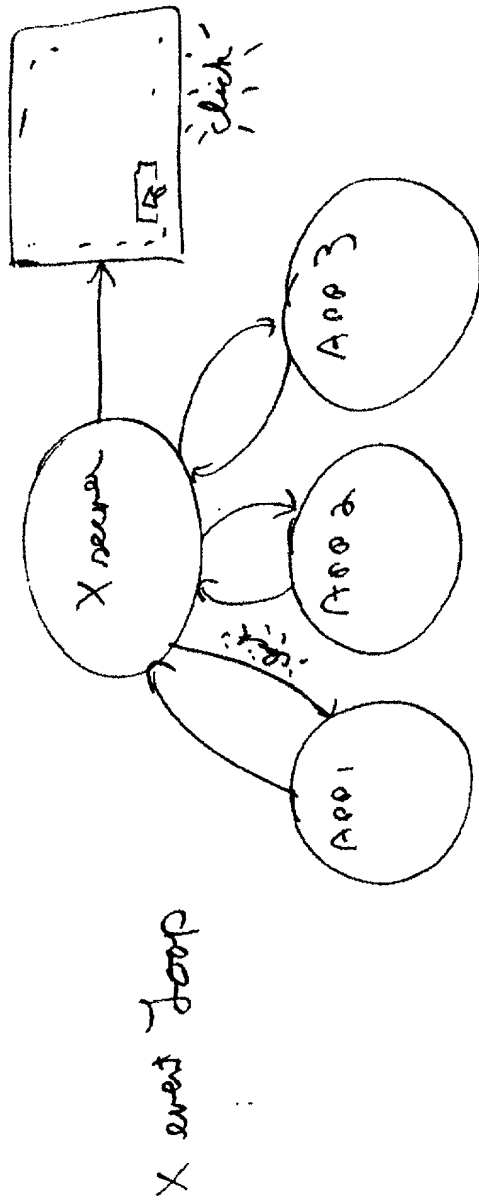


FIG. 6

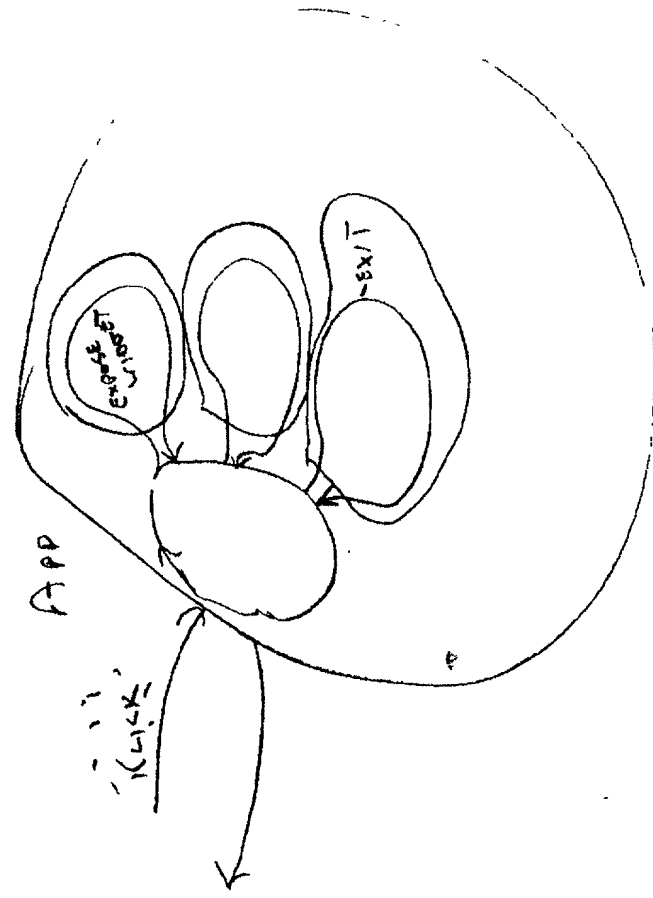
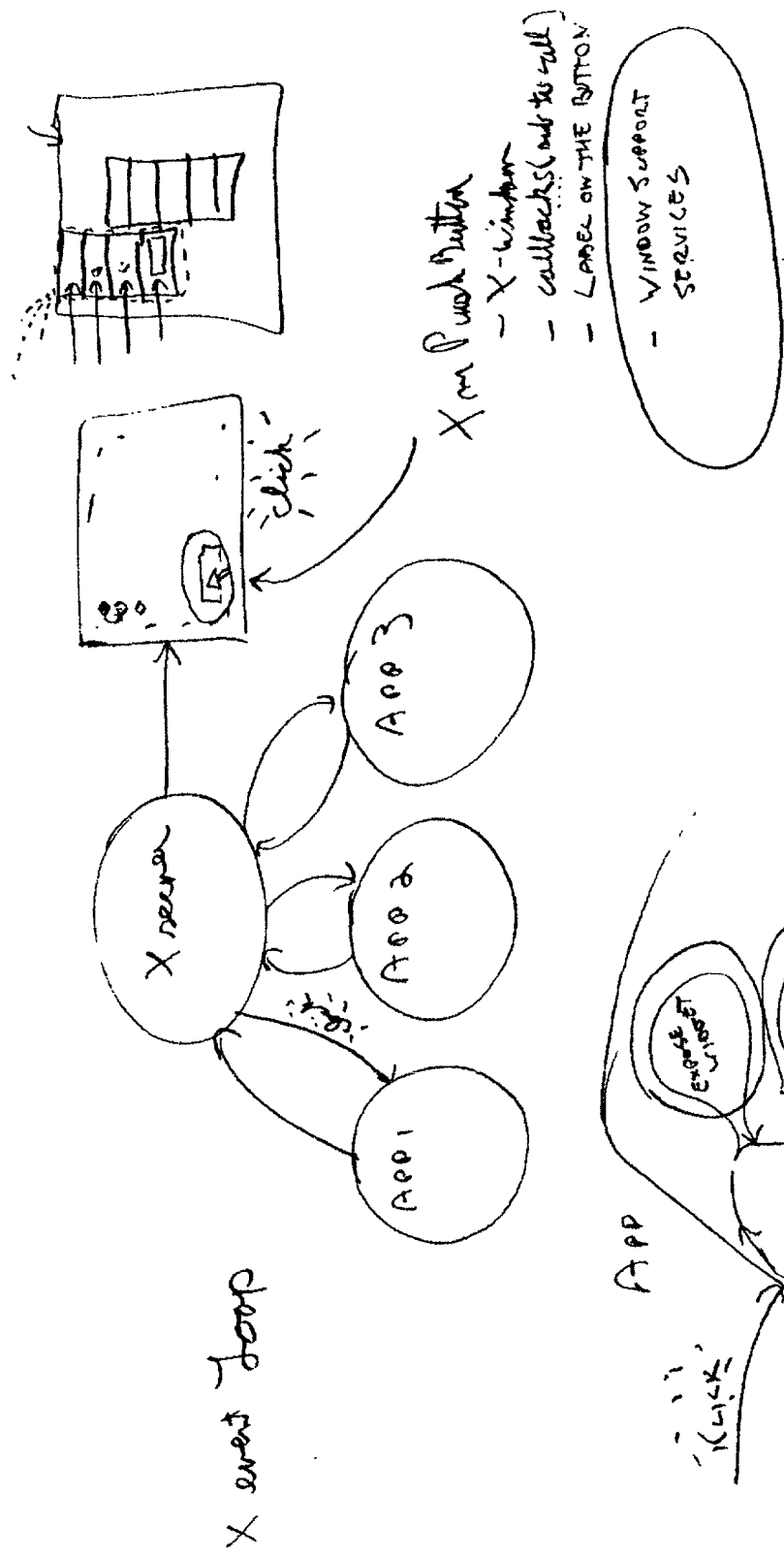


FIG. 7

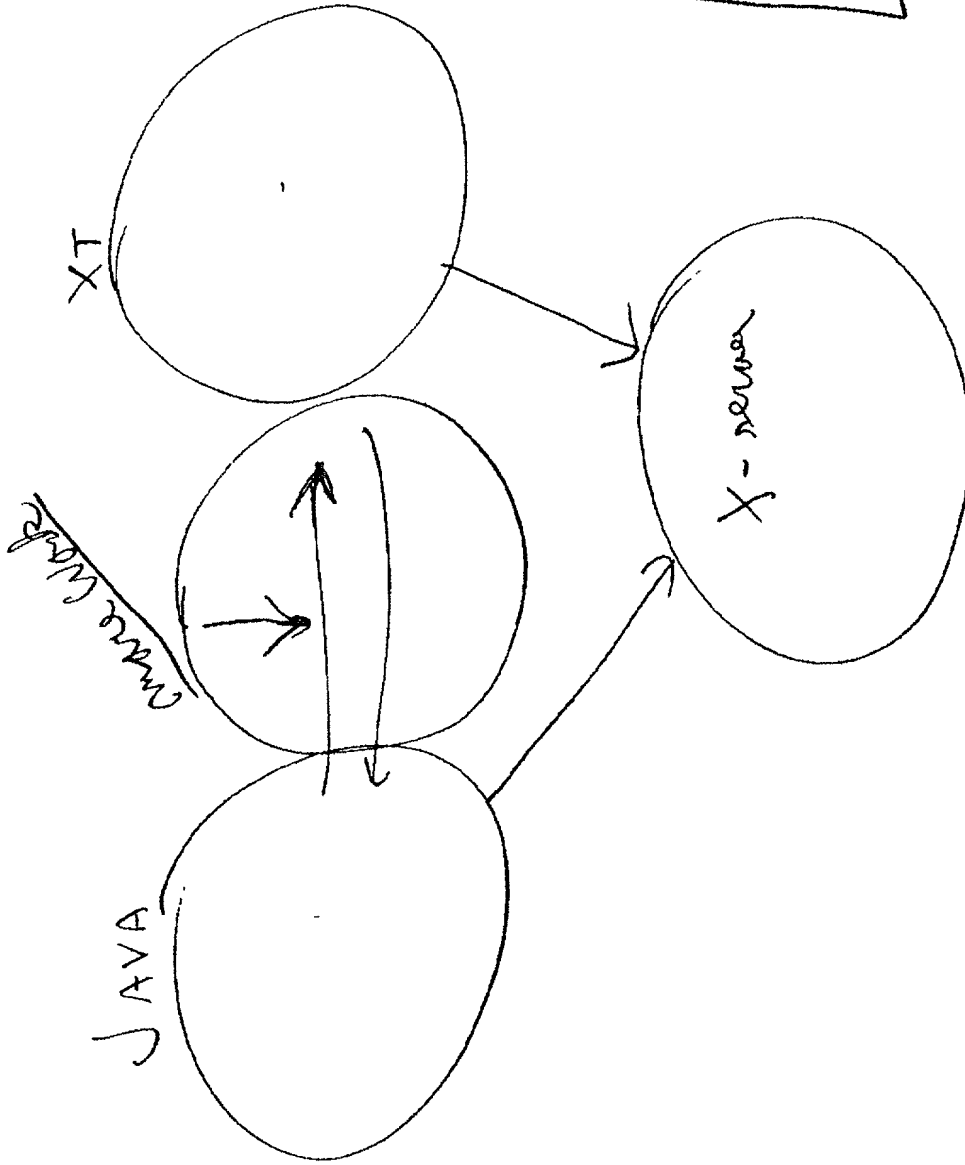
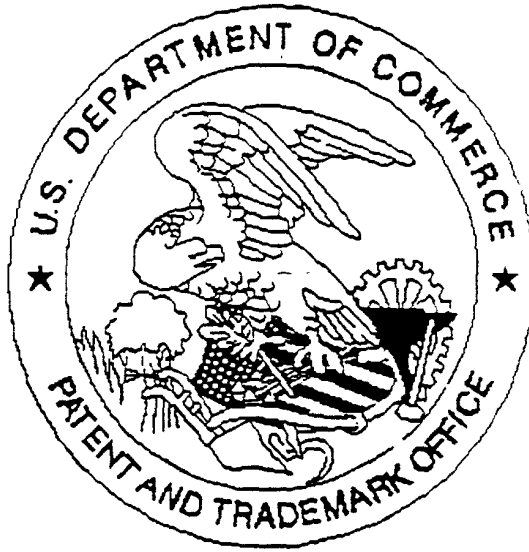


FIG. 8

United States Patent & Trademark Office  
Office of Initial Patent Examination -- Scanning Division



Application deficiencies were found during scanning:

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

☒ Scanned copy is best available. *Appendix A.*